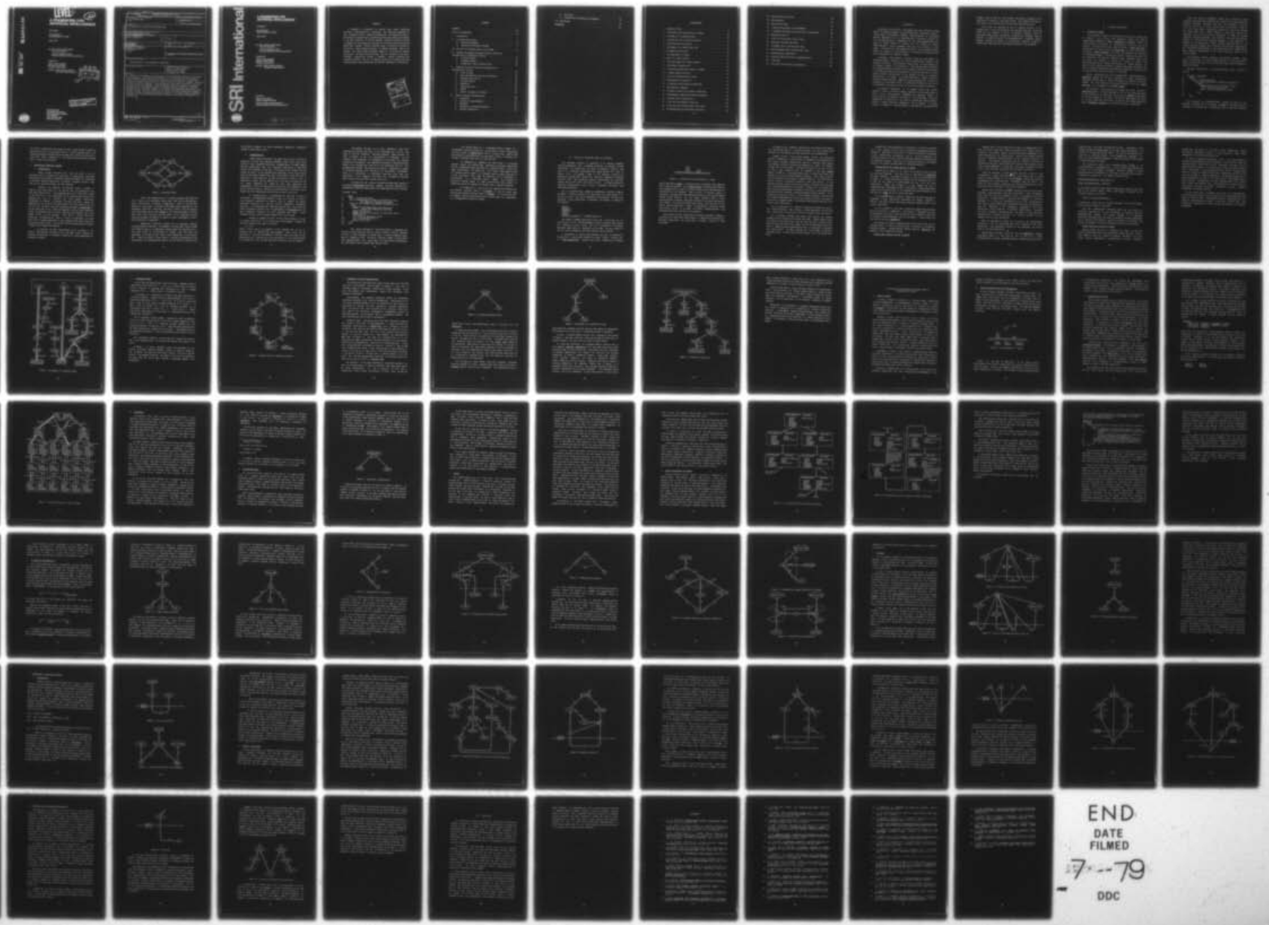

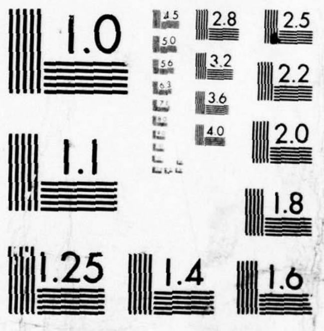
1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

A FRAMEWORK FOR ARTIFICIAL INTELLIGENCE

Final Report

SRI Project 6171
Contract N00014-77-C-0222

March 1979

By: Nils J. Nilsson, Staff Scientist
Principal Investigator

Artificial Intelligence Center
Computer Science and Technology Division

Prepared for:

Office of Naval Research
800 North Quincy Street
Arlington, Virginia 22217

Attention: Marvin Denicoff, Director
Information Systems Branch

SRI International
333 Ravenswood Avenue
Menlo Park, California 94025
(415) 326-6200
Cable: SRI INTL MNP
TWX: 910-373-1246

apr

Security Cl...ation

DOCUMENT CONTROL DATA - R & D

(Securi... ssification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATI... CTIVITY (Corporate author) SRI ...ernational	2a. REPORT SECURITY CLASSIFICATION Unclassified
	2b. GROUP Artificial Intelligence
3. REPORT TITLE (6) A Framework for Artificial Intelligence	
4. DESCRIPTIVE NOTES (Type of report and inclusive dates) (9) Final Reports, March 1979	
5. AUTHOR(S) (First name, middle initial, last name) (10) Nils J. Nilsson	(12) 96p.
6. REPORT DATE (11) March 1979	7a. TOTAL NO. OF PAGES 88 / 7b. NO. OF REFS 48
8a. CONTRACT OR GRANT NO. (15) N00014-77-C-0222 New	9a. ORIGINATOR'S REPORT NUMBER(S)
b. PROJECT NO. 6171	
c.	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
d.	
10. DISTRIBUTION STATEMENT Distribution of this document is unlimited.	
11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Office of Naval Research 800 N. Quincy Street Arlington, VA 22217

13. ABSTRACT

A framework is presented for several major ideas underlying Artificial Intelligence (AI) techniques. Two important components of this framework involve production systems and the predicate calculus. The thesis is proposed that a generalized production system architecture is the basic structure of most AI systems. Predicate calculus provides a language for the production rules and for the data structures on which these rules operate. Several different types of production system are identified; these distinctions help to classify different kinds of AI systems. Two major types correspond roughly to AI systems for theorem proving and for robot problem solving. The last section of the report discusses similarities between a semantic network formalism and predicate calculus. The use of semantic networks in problems involving reasoning about class hierarchies and default information is described.

JOB

DD FORM 1 NOV 65 1473 (PAGE 1)

S/N 0101-807-6801

410 667

Unclassified

Security Classification

SRI International

A FRAMEWORK FOR ARTIFICIAL INTELLIGENCE

Final Report

SRI Project 6171
Contract N00014-77-C-0222

March 1979

By: Nils J. Nilsson, Staff Scientist
Principal Investigator

Artificial Intelligence Center
Computer Science and Technology Division

Prepared for:

Office of Naval Research
800 North Quincy Street
Arlington, Virginia 22217

Attention: Marvin Denicoff, Director
Information Systems Branch

Approved:

Peter E. Hart, Director
Artificial Intelligence Center

David H. Brandin, Executive Director
Computer Science and Technology Division

SRI International

ABSTRACT

A framework is presented for several major ideas underlying Artificial Intelligence (AI) techniques. Two important components of this framework involve production systems and the predicate calculus. The thesis is proposed that a generalized production system architecture is the basic structure of most AI systems. Predicate calculus provides a language for the production rules and for the data structures on which these rules operate. Several different types of production system are identified; these distinctions help to classify different kinds of AI systems. Two major types correspond roughly to AI systems for theorem proving and for robot problem solving. The last section of the report discusses similarities between a semantic network formalism and predicate calculus. The use of semantic networks in problems involving reasoning about class hierarchies and default information is described.

CONTENTS

ILLUSTRATIONS

I Introduction

Researchers in Artificial Intelligence (AI) have developed a large family of computer systems for performing tasks ordinarily thought to require human intelligence. Examples include systems for performing medical diagnoses [1,2], for reasoning about electronic circuits [3], for proving theorems in mathematics [4], for interrogating large data bases to deduce answers to queries [5], for limited but useful understanding of natural language [6-8], and for a variety of other tasks. The people who have designed and built these systems consider them to be based on an evolving body of AI methods and techniques that form a subspecialty of computer science.

To describe their methods, AI people often use such phrases as "knowledge based," "generate and test," "heuristically guided," and "problem reduction." Unfortunately, because many of these phrases are but loosely defined, it is difficult to say precisely what constitutes AI technology without reference to case studies of specific AI programs. To be taken seriously as an engineering or scientific specialty, AI must develop suitable abstractions of its methods that can be analyzed, improved upon, and taught. This report describes a number of basic AI ideas and methods that are common to several applications. We base our discussion on an extended notion of "production systems," as well as on the important role played in AI by the first-order predicate calculus.

A successful abstraction of AI systems should have several features. First, it should be simple, coherent, and well-structured, so that it can be easily taught, remembered, and analyzed. Second, it should capture the essence of the ideas it is abstracting. (Given the abstraction, it should not be too hard to reinvent the actual systems.) Third, it should guide the designer of new AI systems. It should provide a taxonomy of designs matched to applications and promise the designer that he need not work outside the general framework of the abstraction to achieve a successful system. The framework presented in this report is put forward as an initial step toward these goals.

In the next two sections, we give a general overview of our framework and of the role played by the predicate calculus. Following these are two sections in which we describe different types of predicate-calculus-based systems for deduction and planning. Next we show how a currently popular AI representational formalism, namely, semantic networks, is related to the predicate calculus. Finally, we discuss some open questions and problems entailed in the framework.

II AI System Architecture

A. Production Systems

Our framework is based on the observation that most AI systems tend to display a more or less clean separation between the standard computational components of _data_, _operations_, and _control_. That is, if these systems are described at an appropriate level, one can often identify a distinct entity that might be called a _global data base_ -- manipulated by certain well-defined _operations_, all under the control of some global _control strategy_. We stress the importance of identifying an appropriate _level_ of description; near the machine-code level, any neat separation into distinct components might disappear, while at the top level the complete AI system may consist of several data base/operations/control modules interacting in a complex fashion. Our point is that a system consisting of separate data-base, operations, and control components represents an appropriate metaphorical building block for constructing lucid descriptions of AI systems.

Various generalizations of the computational formalism known as a _production system_ [9] involve a separation of these computational components and thus seem to capture the essence of operation of most AI systems. The major elements of a production system in AI are a _global data base_, a set of _rules_, and a _control system_.

The production rules operate on the global data base. Each has a _precondition_ which is either satisfied or not by the global data base. If the precondition is satisfied, the rule can be _applied_. Application of the rule changes the data base. The control system makes the choice as to which applicable rule should be applied and ceases computation when a _termination condition_ on the global data base is satisfied.

There are several differences between this production system structure and conventional computational systems that use hierarchically organized programs. The global data base can be accessed by all the rules; no part of it is local to any of them in particular. Rules do not "call" other rules; communication among rules occurs only through the global data base. These features of production systems are compatible with the evolutionary development of large AI systems requiring extensive knowledge. One difficulty with using conventional systems of hierarchically organized programs is that additions or changes to the system might require extensive modifications of the various existing programs, data structures, and subroutine organization. The production system design is much more modular, and changes can be made relatively independently to the rules to the data base or to the control system.

We distinguish several varieties of production systems. These differ in the kinds of control systems they use, in the properties of their rules and data bases, and in the way in which they are applied to specific problems.

The basic algorithm for a production system can be written as follows:

```
1.   Procedure  PRODUCTION
2.      begin
3.         DATA <- initial data base
4.            until DATA satisfies the termination condition, do:
5.               begin
6.                  Select some rule, R, in the set of rules
                       that can be applied to DATA
7.                  DATA <- result of applying R to DATA

8.               end

9.      end.
```

This procedure is nondeterministic, because we have not yet specified precisely how we are going to select an applicable rule in statement 6. Selecting rules and keeping track of those rule sequences already tried and the states produced by them constitute what we shall call the _control strategy_ for production systems. Efficient control strategies require specialized knowledge about the problem being solved.

We will distinguish two major kinds of control strategies: _irrevocable_ and _tentative._ In an _irrevocable_ control regime, an applicable rule is selected and then applied irrevocably without provision for reconsideration later. In a _tentative_ control regime, an applicable rule is selected, the rule is applied, and provision is made to return later to this point in the computation to apply some alternative rule instead.

We will further distinguish two different types of tentative control regimes. In one, which we call _backtracking_, a _backtracking point_ is established when a rule is selected. Should subsequent computation encounter difficulty in producing a solution, the state of the computation reverts to the state at a previous backtracking point by undoing the effects of all intervening computation. At this point another rule is invoked and the process continues _as if the first rule had never been applied at all_.

In the second type of tentative control regime, which we call _tree-search control_, provision is made for keeping track of the effects of several sequences of rules simultaneously. Various kinds of tree structures and tree-searching procedures are used in this type of control. (An irrevocable control regime corresponds to following just a single path down through the tree. A backtracking regime does not maintain the entire tree structure; it merely keeps track of the path it is working on currently and modifies it when necessary.)

The problem faced by a tree-search control regime for a production system can be described as the problem of searching a graph. Of special interest are graph-searching methods that use what is called _heuristic knowledge_ from the problem domain to focus the search toward producing a global data base that satisfies the termination condition. (Heuristic graph-searching methods are described in detail in Nilsson [10]. Although these methods play an important role in AI systems, they are by now well understood and are consequently not treated in this report.)

Many problems to which production systems are applied involve changing a given problem situation into one satisfying some goal condition. For these kinds of problems we might represent the problem situation by a formal structure called a _state description_ and use it as the global data base of a production system. The termination condition of the production system then involves a test to see if the global data base satisfies the goal condition of the problem. We shall say that such a production system works on a problem in the _forward direction_.

Alternatively, we might represent the goal condition of a problem by a formal structure called a _goal_ and use this goal as the global data base of a production system. The production rules applied to the global data base then create subgoals. The termination condition of the production system then involves a test to see if the initial problem situation satisfies the goal expressed by the global data base. Such a production system works on a problem in the _backward direction_.

Although there is no formal difference between a production system that works on a problem in a forward direction and one that works in a backward direction, it is often convenient to make this distinction explicit. Rules that are applied to state descriptions to produce new state descriptions will be called _F-rules_; those that are applied to goals to produce subgoals will be called _B-rules_. Tree-search control regimes for forward systems generate a tree of state descriptions; tree-search control regimes for backward systems generate a tree of goal descriptions.

A solution to a problem can be attempted in both directions simultaneously. We can achieve this effect with production systems also. To do so, we must incorporate both state descriptions and goal descriptions into the global data base. F-rules are applied to the state description part only, while B-rules are applied to the goal description part only. The termination condition to be used by the control system to decide when the problem is solved must now be represented as some type of matching condition between the state description part and the goal description part of the global data base.

The control system must also decide at every stage whether to apply an applicable F-rule or an applicable B-rule. In many problems there is an underlying dualism between states and goals and between F-rules and B-rules that leads to symmetries in the bidirectional production systems used to solve these problems.

B. Specialized Production Systems

1. Commutativity

Under certain conditions the order in which a set of applicable rules is applied to a data base is unimportant. When these conditions are satisfied, a production system can improve its efficiency by avoiding needless exploration of several redundant solution paths that are all equivalent except for rule ordering.

Consider, for example, the situation shown in Figure 1. There we have three rules, R1, R2, and R3, that are applicable to the data base denoted by S0. After applying any one of these rules, either of the other two is still applicable to the resulting data bases; after applying any pair in sequence, the third remains applicable. Furthermore, in Figure 1 we produce the same data base, namely SG, regardless of how we order the application of the three rules in the set {R1, R2, R3}. We shall say that a production system is commutative if, for any data base, we can apply any sequence of rules from an arbitrary subset of applicable rules to yield a data base that depends only on the subset and not on the sequence. Note that it follows from this definition of commutativity that, if a rule is applicable to a data base, then it remains applicable to any subsequent data base produced by applying any other rule(s). We see that the rule applications in Figure 1 possess this commutative property.

In producing the data base denoted by SG in Figure 1, we clearly need consider only one of the six paths shown. Methods for avoiding consideration of redundant paths are of great importance for commutative systems.

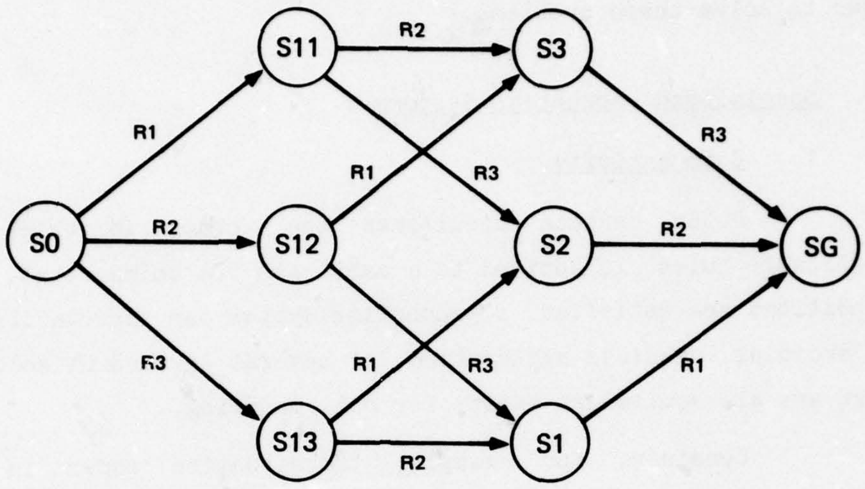

Figure 1 Equivalent Paths

(Note that commutativity of a system does not mean that the set of rules used to transform a given data base into one satisfying a given condition can be applied in any order. After a rule has been applied to a data base, additional rules might become applicable to the subsequent data base. Permuting the order in which rules are applied can be performed only for that subset of the rules applicable to a given data base. Only the rules in that subset can be organized into any arbitrary sequence and applied to produce a result independent of order. This distinction is important.)

Commutative production systems are an important subclass enjoying special properties. For example, an irrevocable control regime can always be used in a commutative system because the application of a rule never needs to be rescinded or undone. Any rule that was applicable to an earlier data base is still applicable to the current one. There is no need to keep track of alternative sequences of rule applications; an inappropriately applied rule can easily be removed from the solution sequence. We shall investigate examples of commutative systems in more detail later.

2. <u>Decomposability</u>

Under certain conditions, the global data base of a production system can be split into separate components, after which rules can be applied to each component separately. Metaphorically, we might imagine that the global data base is a "molecule" consisting of individual "atoms" bound together in some way. (The nature of the "bonds" need not concern us now.) If the applicability conditions of the rules involve tests on individual "atoms" only, and if the effects of the rules are to substitute a qualifying atom by some new molecule (that in turn is composed of atoms), then we might as well have split the molecule into its atomic components and worked on each part separately and independently. Each rule application affects only that component of the global data base used to establish the precondition of the rule, and we assume that this transformed part can also be split, and so on.

To accomplish such a decomposition we must also be able to "decompose" the termination condition. That is, if we are to work on each component separately we must be able to express the global termination condition as a function of each component's termination condition. An interesting and important special case occurs when the global termination condition can be expressed as the <u>conjunction</u> of the same termination condition for each component data base. Unless otherwise stated, we shall always assume this to be the case.

Consider an example in which the data base consists of a list of symbols, such as (C, B, Z), drawn from some alphabet. Suppose the rules are rewrite rules, each in the form

X --> {Yi}

where X can be any of the symbols of the alphabet, and {Yi} is an arbitrary list of symbols from that alphabet. Such a rule can be applied if X is one of the symbols in the data base; the effect of the application is to replace one of the occurrences of X in the data base by the symbols in {Yi}, so that the result is still a list of symbols.

Now suppose our goal is to find a sequence of rules that transforms the initial data base to one containing, say, all M's. Here again we have the problem of exploring redundant paths, because the order of several of the rule applications is irrelevant. We can eliminate such redundancy by decomposing or splitting the original problem into the separate and independent problems of transforming each symbol in the original data base into a list containing only M's. This decomposition involves the separate and parallel operation of a number of production systems, each working on its own problem. Furthermore, there is a possibility of additional splitting after each rule application. In our example a rule adds a list of symbols to the data base; this new list can also be split, and so on.

We shall call the class of production systems that operate in this manner _decomposable_ production systems. The basic procedure for a decomposable production system might look something like the following:

Procedure SPLIT

```
1.   begin
2.     DATA <= initial data base
3.     {Di} <= Decomposition of DATA; the individual Di are
                now regarded as separate data bases
4.     Until all {Di} satisfy the termination condition, do:
5.       begin
6.         Select D* from among those {Di} that do not
                satisfy the termination condition.
7.         Remove D* from {Di}.
8.         Select some rule R in the set of rules that can be
           applied to D*.
9.         D <= result of applying R to D*.
10.        {di} <= Decomposition of D.
11.        Append {di} to {Di}
12.      end
13.  end
```

The control strategy for SPLIT must select a component data base D* in Step 6 and must select a rule R to apply in Step 8. Whatever the form of this strategy, to satisfy Step 4 it must ultimately select _all_ the elements in {Di}. For any D* selected, though, it need select only _one_ applicable rule. Again there are irrevocable, backtracking, and tree-search control regimes that might be used.

The problem faced by a tree-search control regime for a decomposable production system can be described in terms of searching a structure called an AND/OR tree to find a solution tree. AND/OR trees and methods for searching them are described in detail in Nilsson [10]. We shall illustrate their use in a later section.

To summarize, we recognize two major types of production systems: namely, the ordinary type (described by procedure PRODUCTION) and the decomposable type (described by procedure SPLIT). Either type might, additionally, be commutative. Any of these types might be used in a forward (state to goal) or a backward (goal to state) direction or in both directions. They might be controlled by an irrevocable, a backtracking, or a tree-seach control regime. The taxonomy of production systems based on these distinctions helps greatly in organizing various AI systems and concepts into a coherent framework.

It is important to note that we are drawing distinctions here only between different kinds of AI systems; we are not making any distinctions between different kinds of problems. It is often the case, for example, that the same problem can be solved either by a commutative or a noncommutative production system.

III The Role of Predicate Logic in AI Systems

The predicate calculus is important in AI because predicate calculus expressions are a convenient formalism for encoding problem information. (We assume that the reader is generally familiar with logic. For a readable introduction see pages 109-138 of a book by Raphael [11].) AI systems based on the predicate calculus are production systems whose global data bases are composed of predicate calculus wff's. The production rules of such systems are rules that change sets of wff's into other sets of wff's. In typical operation, rules are applied until a set of wff's is produced that meets some termination condition.

Let us illustrate this scheme by considering a specific example. In Figure 2 we show a situation in which there are three blocks, A, B, and C, on a table. How can we represent this situation? One way is to represent it by a conjunction of formulas. For example, we might use the formulas:

```
ON(C,A)
ONTABLE(A)
ONTABLE(B)
CLEAR(C)
CLEAR(B)
(FOR-ALL x)[CLEAR(x) => ~(EXISTS y)ON(y,x)]
```

The formula CLEAR(B) means that block B has a clear top; i.e., no other block is on it. The ON predicate is used to describe which blocks are (directly) on other blocks. (ON is not transitive.) The formula ONTABLE(B) means that B is somewhere on the table. The last formula in the list gives information about how CLEAR and ON are related.

A conjunction of several such formulas can then be considered a description of a particular situation or "world state." We shall call it a <u>state description</u>. Actually, any finite conjunction of formulas

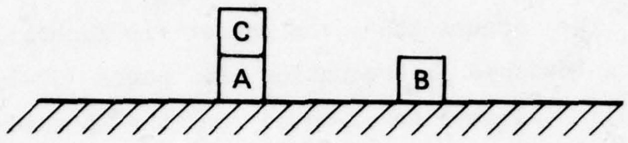

Figure 2 A Situation with Three Blocks on a Table

really describes a _family_ of different world situations, each member of which might be regarded as an _interpretation_ satisfying the formulas. Even assuming that we give the obvious "blocks-world" interpretation to constituents of the formulas, there is still an infinite family of situations (perhaps involving additional blocks as well) whose members satisfy these formulas. We can always eliminate some of these interpretations by adding additional formulas to the state description. (For example, the set listed above says nothing about the color of the blocks and thus describes a family of situations in which the blocks have various colors. If we added the formula COLOR(B,YELLOW), some interpretations would obviously be eliminated.)

Even though a finite conjunction of formulas describes a family of situations, we shall often speak loosely of _the_ situation described by the state description. We really mean, of course, the set of such situations.

We intend to use formulas, like those of our blocks-world example, as a global data base in a production system. The way in which these formulas are used depends upon the problem and its representation.

Suppose we have a "robot-type" problem in which the system must specify a sequence of robot actions for changing the configuration of the blocks. We might describe the goal by a predicate logic formula that defines the set of situations acceptable as a goal. For example, we might want to have block A on block B and block B, in turn, on block C. Such a goal situation could be expressed by the goal formula [ON(A,B) & ON(B,C)]. Or we might merely want to verify that block B has nothing on it in the situation shown in Figure 2. Such a goal situation could be described by the formula ~(EXISTS x)ON(x,B). In either case we want to specify a production system to solve the problem. If the production system were to work in the forward direction, we would take the formulas describing the situation of Figure 2 as the global data base (the state description) and apply F-rules until a state description was produced that contained a formula matching the goal formula. If the production system were to work in the backward direction, we would take the formula describing the goal as the global data base and apply B-rules until a subgoal formula was produced that matched formulas in the initial state description.

Let us concentrate for a moment on production systems that work in the forward direction. Here we have a state description as the data base. What kinds of F-rules can be used to transform this description?

For robot-type problems the F-rules must specify the precise changes that can be made to the global data base. Usually these F-rules will model the various actions that the robot can take. When the robot takes an action its world will change, and the corresponding F-rule will have to change the state description so that it continues to represent the robot's world. In other problems the F-rules might be based on inference rules, in which case they will merely add to the global data base.

Suppose the state description (a conjunction of wff's) is denoted by S. Let <u>S</u> stand for the set of situations (interpretations) that satisfy S. Application of an F-rule to a state description S transforms it into another state description, say, S'. We can categorize the different types of F-rules used with predicate logic state descriptions by comparing <u>S</u> and <u>S'</u>. There are two major types that will correspond nicely to our earlier classification of production systems.

<u>F-Rules That Do Not Change The Set of States</u>

Suppose <u>S'</u> is a subset of <u>S</u>. First, let us take the case in which <u>S'</u> and <u>S</u> are identical. Here the F-rule is used merely to <u>redescribe</u> the same situation. An example of such an F-rule is an inference rule of logic. If a state description contains the two formulas P and P => Q, then we can infer Q by modus ponens and add it to the state description. Adding this formula in no way changes the set of interpretations satisfying the state description. Incorporating modus ponens into an F-rule allows us to elaborate the state description; its use does not change the set of states described.

In our example shown in Figure 2, suppose we were given the goal of producing a state description containing the formula ~(EXISTS x) ON(x,B). This goal could be achieved by applying inference rules to the initial description.

It is of interest to note that a production system using inference rules as F-rules is commutative. (Any subset of inference rules that are applicable to a given state description can be applied in any order to produce a result depending only on the subset.) Thus we shall say that such F-rules are <u>commutative</u>.

The problem of proving a theorem, given a set of wff's and a set of inference rules, is thus a problem that can be solved by a commutative production system. Theorem-proving systems based on <u>resolution</u> are examples of such commutative systems.

<u>F-Rules that Restrict the Set of States</u>

Another type of F-rule produces an S' that is a proper subset of S. Then the F-rule must have added new formulas that further specify a situation. For example, suppose we added the formula ONTABLE(D) to the state description of our example above. Clearly, this addition narrows the set of possible worlds described by the original conjunction of formulas. (The original description did not preclude a block D from entering into diverse relationships with various other possible blocks. After adding ONTABLE(D), we know that only those situations in which block D is on the table can satisfy the new state description.)

There are two ways in which we might add such a piece of new information to a state description. The most obvious one, in robot problem solving systems, is by a direct sensory action that discovers supplementary and uncontradictory information about the world. The effect of such a sensory perception could then be achieved by an F-rule that added the new statement to the state description.

Another way to add more statements involves the use of F-rules that represent certain general facts about situations. Thus, for example, rather than have both the statements P and P => Q contained in the state description, we might choose to have only the statement P. The other statement P => Q could be incorporated in an F-rule. Application of this rule to a state description containing P would add the statement Q. From the standpoint of the information contained in the state description, such an addition is a further specification. It narrows the set of situations being described. The source of this additional information was of course the rule P => Q, which we chose to keep separate from the state information.

When an implication is used as an F-rule, its applicability condition is that its antecedent must match formulas in the state description. The appropriate instance of the consequent can then be added to the state description.

Implicational F-rules of this type are also commutative. Another interesting feature of this kind of rule is that it is rather straightforward to design backward production systems whose B-rules are simply derived from these implicational F-rules. Furthermore, these backward systems are decomposable (or at least almost decomposable -- in a sense to be made more precise later). The use of implications as F-rules was an important feature of the PLANNER-like languages [12]. Moore [13] gives detailed arguments to support the value of this feature in enhancing system efficiency.

One obvious and direct use of theorem-proving systems is for proving theorems in mathematics [4]. A less obvious but important use of them is in intelligent information retrieval systems in which deductions must be performed on a data base of facts in order to derive an answer to a query [14]. For example, from facts like

MANAGER(PURCHASING-DEPT, JOHN-JONES) and

WORKS-IN(PURCHASING-DEPT, JOE-SMITH)

an intelligent retrieval system might be expected to answer a query like "Who is Joe Smith's boss?" Such a query might be stated as the following theorem to be proved:

(EXISTS x) BOSS-OF(JOE-SMITH,x).

A constructive proof (that is, one that exhibited the "x" that exists) would provide an answer to the query.

Even many "common-sense" reasoning tasks that one would not ordinarily formalize can in fact be handled by predicate calculus theorem-proving systems. The general strategy is to represent specialized knowledge about the domain as predicate calculus expressions and to represent the problem or query as a theorem to be proved. The system then attempts to prove the theorem from the given expressions.

F-Rules That Shift The Set of States

Suppose neither S' nor S is a subset of the other. In this case the F-rule has shifted the set of situations to some other set. Such F-rules might be used, for example, to describe the result of a robot action, such as rearranging a configuration of blocks. Production systems that use F-rules of this type are not commutative, because applying one applicable F-rule to a state description might render inapplicable an otherwise applicable rule.

We shall be especially interested in two of the above classes of predicate-calculus-based production systems. In the first of these, the forward-directed version will have as its global data base a set of predicate-logic literals. Its F-rules will be based on simple predicate logic implications. In the backward-directed version, the global data base will consist of predicate-logic goal expressions, and the B-rules will also be based on simple implications. The backward system will be almost decomposable, so that it will be possible to use AND/OR trees to control search. Although some of these systems suffer from certain minor logical deficiencies, they are nevertheless of great utility in applications involving intelligent information retrieval, automatic programming, and theorem proving.

In the second class of systems the F-rules are noncommutative. Backward-directed versions of such systems are not decomposable and thus cannot use AND/OR trees. Nevertheless, several techniques have been developed employing methods related to the use of AND/OR trees. Such systems find application in robot problem solving and in automatic programming.

IV Forward and Backward Predicate Calculus Production Systems Based on Commutative F-Rules

A. A Forward System

We first consider some systems for proving a goal formula from a set of assertions. For the sake of simplicity we shall limit our discussion here to the use of a restricted class of logical formulas. The goal formula can consist of a conjunction of literals. The assertions can be either single literals called facts or simple implications. These implications can have a conjunction of literals as their antecedent and a single literal as their consequent. Furthermore, we assume that all variables occurring in facts and implications are universally quantified and that all variables in goals are existentially quantified. (Existential variables in facts and implications and universal variables in goals can be Skolemized, so that the resulting expressions conform to our assumptions.)

We first consider a forward commutative production system whose global data base consists of the facts and whose rules are derived from the implications. As an example, consider the following facts:

```
F1:  DOG(FIDO)
F2:  ~BARKS(FIDO)
F3:  WAGS-TAIL(FIDO)
F4:  MEOWS(MYRTLE)
```

and the following rules:

```
R1:  [WAGS-TAIL(x1) & DOG(x1)] => FRIENDLY(x1)
R2:  [FRIENDLY(x2) & ~BARKS(x2)] => ~AFRAID(y2, x2)
R3:  DOG(x3) => ANIMAL(x3)
R4:  CAT(x4) => ANIMAL(x4)
R5:  MEOWS(x5) => CAT(x5)
```

We use the convention that variable symbols are represented by strings beginning with lowercase letters near the end of the alphabet

(e. g., ..., x, y, z). Constant symbols and predicate symbols are represented by strings of uppercase letters. Function symbols are represented by strings of lowercase letters. We standardize variables apart in the rules to avoid confusion during substitutions.

Suppose we are trying to conclude that there are a cat and a dog, and that the cat is unafraid of the dog. The goal expression is then:

[CAT(x) & DOG(y) & ~AFRAID(x,y)]

We say that an implicational F-rule is applicable to the set of facts if there exists a substitution _s_ that simultaneously unifies each of the conjuncts in the antecedent of the rule with one of the facts. (See [15] for a discussion of most general simultaneous unifiers.) We then say that the antecedent of the rule _matches_ the facts. To apply an applicable rule, we apply the substitution _s_ to the rule consequent and add this instance to the set of facts. The termination condition for this production system is met when there is a substitution that simultaneously unifies each of the goal conjuncts with a fact (that is, when the goal matches the facts.)

In Figure 3 a _deduction graph_ shows how the F-rules are used to produce a solution to our sample problem. Nodes connected by double lines are unifiable expressions; the double lines are labeled by the corresponding most general unifier (mgu). The doubly boxed expressions are the original facts, while the singly boxed nodes constitute the goal expression. They match (unify with) facts in the deduction graph as shown in the figure, and the simultaneous unifier of the match is {MYRTLE/x, FIDO/y, MYRTLE/y2}. When this substitution is applied to the goal expression we obtain an _answer statement_ [CAT(MYRTLE) & DOG(FIDO) & ~AFRAID(MYRTLE, FIDO)]. A "proof" of this statement is shown by the darkened branches of the graph.

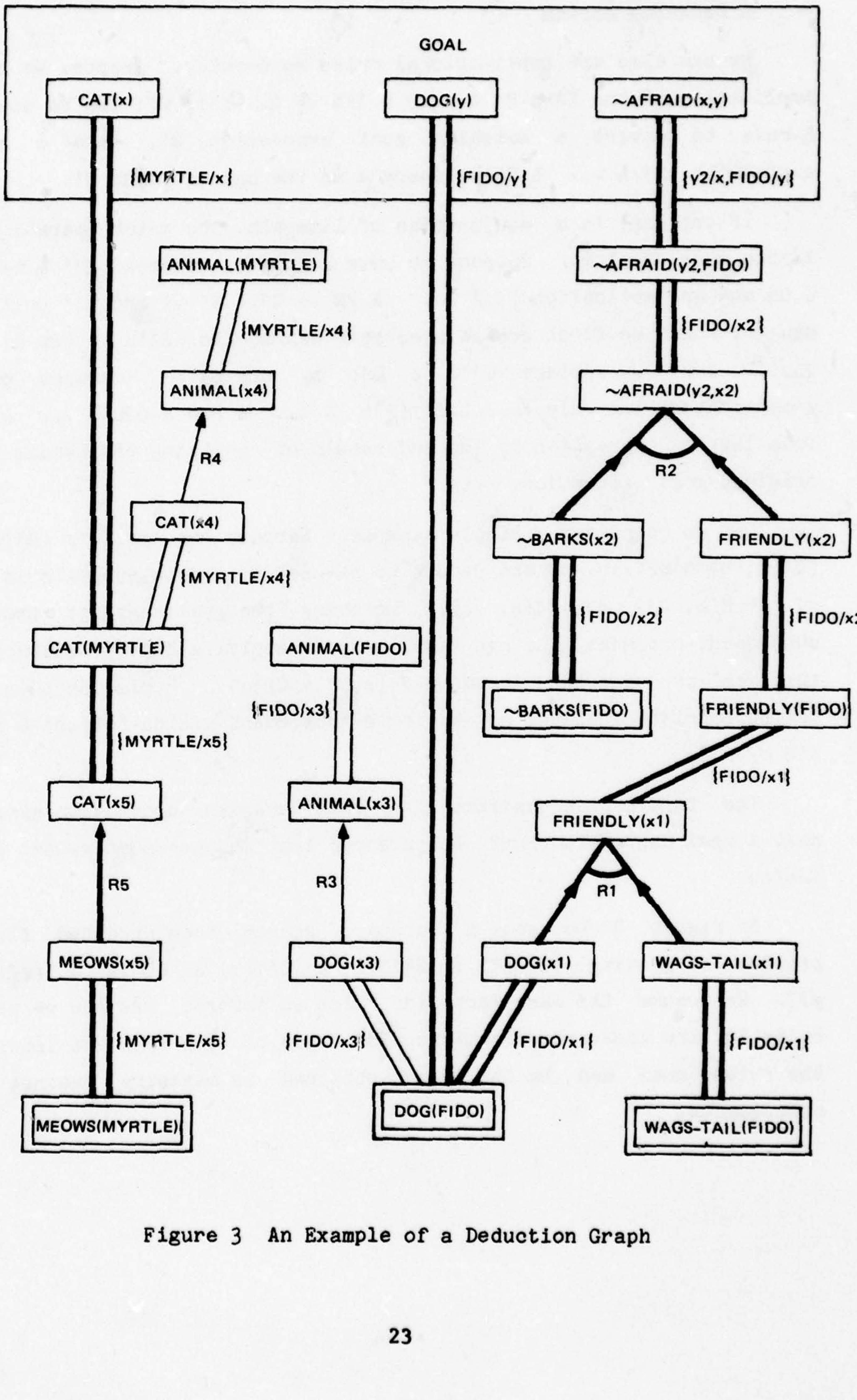

Figure 3 An Example of a Deduction Graph

B. A Backward System

We can also use implicational rules as B-rules. Suppose we have an implication of the form P1 & ... & PN => Q. This wff can be used as a B-rule to convert a matching goal expression, Q', into a subgoal expression (P1 & ... & PN)u, where u is the mgu of Q and Q'.

If the goal is a conjunction of literals, the match operation is a little more complex. Suppose we have a goal expression Q1' & Q2 & ... & QN and an implication P1 & ... & PN => Q1. If Q1 and Q1' unify with mgu, u, then we first create a subgoal expression (Q1'u & Q2u & ... & QNu). Next we replace Q1'u = Q1u by the match instance of the antecedent of the rule to obtain (P1u & ... & PNu & Q2u & ... & QNu). This latter expression is the end result of applying the B-rule to the original goal expression.

Let us consider a simple example. Suppose the goal is (EXISTS z) [Q1(z, b) & Q2(z)] and the B-rule to be used to create subgoals is [P(y, u) & R(a, u)] => Q1(a, u). Matching the goal against the rule consequent produces the mgu {a/z, b/u}. Applying this substitution to the goal produces the instance Q1(a,b) & Q2(a). Replacing Q1(a,b) by the appropriate instance of the rule antecedent yields [P(y,b) & R(a,b) and Q2(a)].

The termination condition for this backward production system is that a goal expression must be produced that matches some subset of the facts.

In Figure 4 we show a backward search tree produced for the problem of proving (EXISTS x)(EXISTS y)[CAT(x) & DOG(y) & ~AFRAID(x, y)]. We assume the same facts and rules as before, but now we use the rules in the backward direction. The edges of the tree are labeled by the rules used and by the mgus obtained in matching against rule consequents.

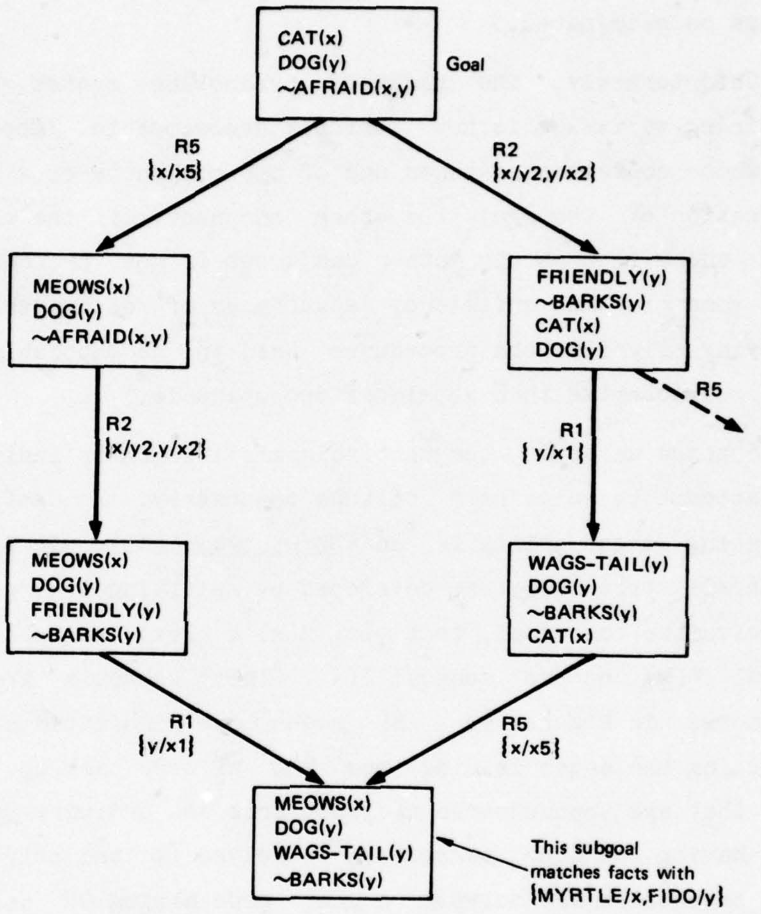

Figure 4 A Search Tree for a Deduction Problem

C. Attempting to Split Compound Goals

The problem just considered had a small search space, but note that there are two equivalent solution paths in Figure 4. This sort of redundancy might be intolerably inefficient in larger problems. If the goal and subgoals were decomposable, however, the redundancy could perhaps be eliminated.

Unfortunately, the backward production system for expressions containing variables is not strictly decomposable. Application of a B-rule whose consequent matches one of the conjuncts of a goal affects (by application of the mgu) the other conjuncts of the goal expression. Since the effect on the other conjuncts is not profound, we can still gain some of the efficiency advantages of decomposable systems by modifying slightly the procedures used in decomposable systems. (We might characterize them as almost decomposable.)

Suppose we split conjunctive goals into their individual conjuncts and attempt to solve each of them separately. A useful structure for indicating these splits is an AND/OR tree [10]. In Figure 5 we show the AND/OR tree structure developed by splitting the goal $P(x)$ & $Q(x)$. To solve the top goal, that is, $P(x)$ & $Q(x)$, we must solve both the subgoal $P(x)$ and the subgoal $Q(x)$. These subgoals are represented by AND nodes in Figure 5. AND nodes are indicated such by an arc connecting the edges leading from the AND node back up to their parent. Nodes that are unencumbered by such arcs are ordinary OR nodes. A goal node having AND node successors is solved if and only if all its AND node successors are solved; a goal node having OR node successors is solved if and only if at least one of its OR node successors is solved. A tip node is solved if the goal that it denotes matches one of the facts. The tree that demonstrates (according to the above definition) that the top node is solved is a solution tree.

The occurrence of the same existentially quantified variable x in each of the split goals in Figure 5 indicates that these goals cannot be solved independently. If, say, a solution for $P(x)$ uses a substitution $\{a/x\}$, so must the solution for $Q(x)$. Thus, we can use

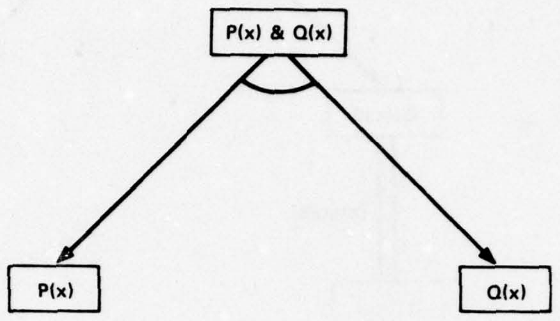

Figure 5 Splitting Conjunctive Goals

AND/OR trees only if the substitutions used in a solution tree are consistent.

To show how and an AND/OR tree is developed by the use of B-rules, suppose we have the goal Q1(z,b) & Q2(z) and the B-rule P(y,u) & R(a,u) => Q1(a,u). First, we split the goal into its individual conjuncts and then we can apply the rule to one of them. The AND/OR tree structure thus created is shown in Figure 6. We show a match edge (labeled by the appropriate substitution) linking the rule conjunct to the goal to which it is applied; the appropriate instance of the rule antecedent is then split, and the components are shown as AND nodes in the tree. In the tree of Figure 6 the variable y may have anything substituted for it in proving the component P(y,b), but the variable z must have "a" substituted for it in proving Q2(z).

One strategy for using AND/OR trees with backward, rule-based deduction systems is to split compound goals and to apply rules until a candidate AND/OR solution tree is produced. The substitutions used in

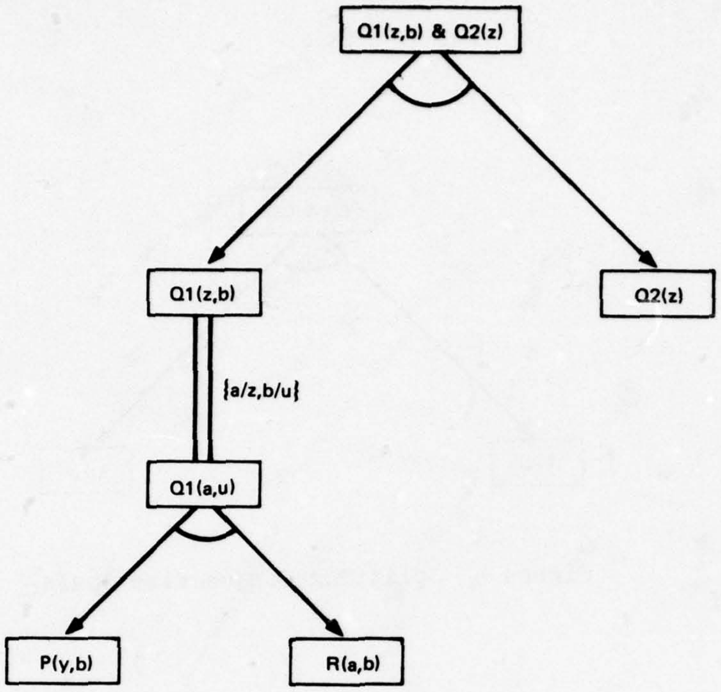

Figure 6 Development of an AND/OR Goal Tree

developing this candidate solution tree are then tested for consistency. The problem is not solved until a consistent solution is produced.

We show a candidate AND/OR solution tree produced by this process in Figure 7. The goal, rules and facts are the same as those used in a previous example. The fact nodes are shown in double boxes.

To check the consistency of a candidate AND/OR solution tree, we must compute a unifying composition of all the substitutions that label the match edges of the solution tree. (See Sickel [16] for a discussion of unifying compositions.) For Figure 7 we must find the unifying composition of ({x/x5}, {MYRTLE/x}, {FIDO/y}, {x/y2, y/x2}, {FIDO/y}, {y/x1}, {FIDO/y}, {FIDO/y}). The result is {MYRTLE/x5, MYRTLE/x, FIDO/y, MYRTLE/y2, FIDO/x2, FIDO/x1}. Because we were able to compute a unifying composition, the candidate AND/OR tree of Figure 7 is consistent and we have a solution. This unifying composition applied to the goal expression yields an answer statement. If a candidate AND/OR solution tree is not consistent, we must then continue to apply rules

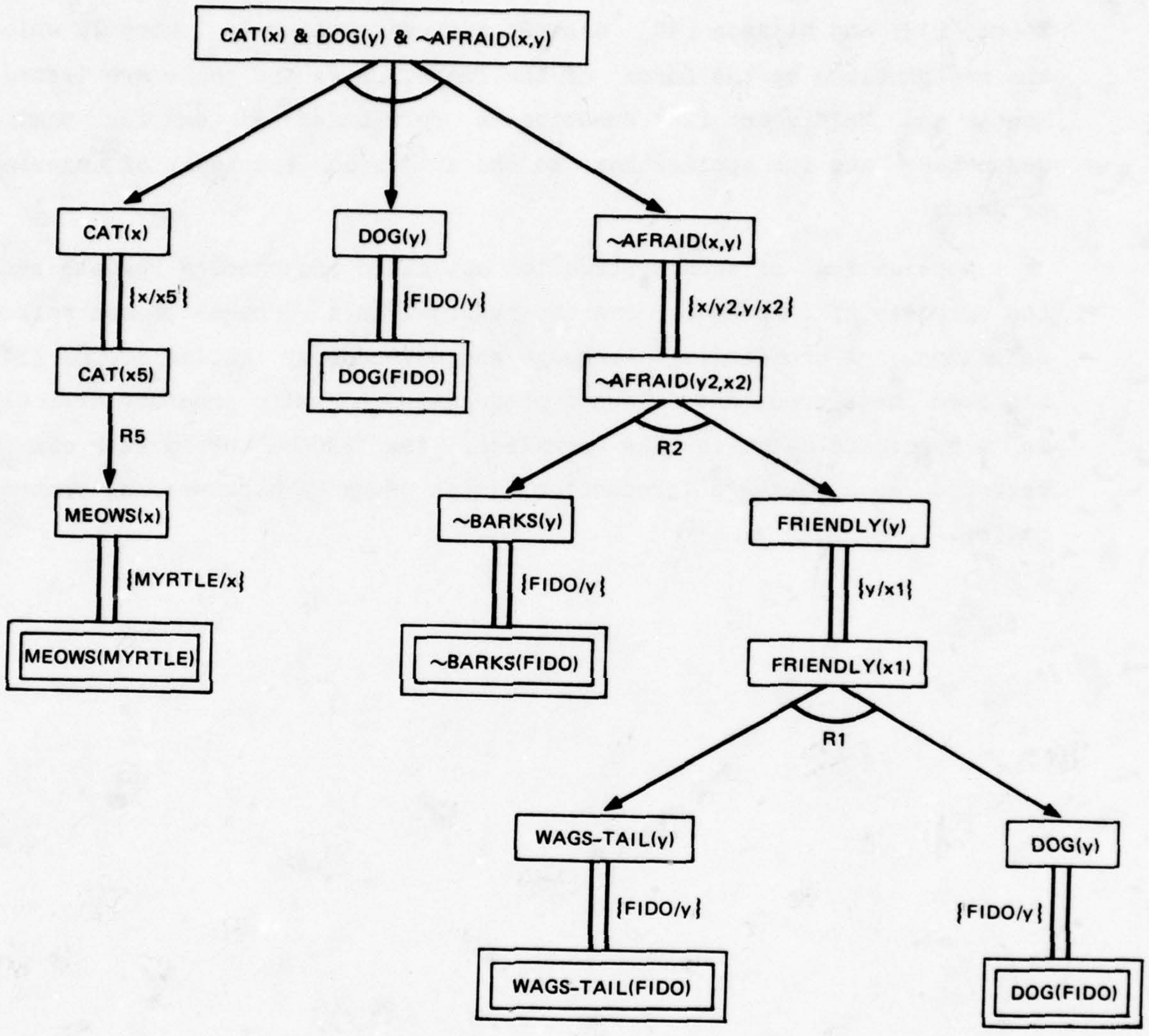

Figure 7 An AND/OR Solution Tree

until another candidate is found, and so on. The technique of first finding candidate solutions and then checking them by computing unifying compositions is similar to that of Klahr [17] and Cox [18].

There are many ways in which the above systems can be generalized. Moore [13] and Nilsson [19] discuss systems similar to these in which the restrictions on the forms of the facts, rules and goals are lifted. Manna and Waldinger [20] describe a rule-based system for making deductions and its application to the automatic synthesis of computer programs.

Application of such systems to automatic programming reveals that the process of computation can be regarded as a process of controlled deduction. A programming language and methodology called PROLOG [21] has been developed that allows a programmer to write programs directly in a predicate-calculus-like formalism. The PROLOG interpreter can be regarded as a backward production system using a backtracking control regime.

V Predicate Calculus Production Systems Based on Noncommutative F-Rules

A. Robot Problems

For many problems of interest in AI the most natural formulations involve noncommutative or nondecomposable systems. Typical of these kinds of problems are those in which a goal is achieved by a sequence (or program) of actions that are best modeled by noncommutative F-rules. Robot problem solving and automatic programming are two domains in which such problems occur.

Research on robot problem solving has led to many of our ideas about problem-solving systems. Since they are simple and intuitive, we shall use examples of robot problems to illustrate the major ideas. In the typical formulation of a "robot problem" we are to imagine that a robot has a repertoire of primitive actions that it can perform in some easily comprehensible world. In the "blocks world," for example, we imagine that there are several labeled blocks resting on a table or on one another. The robot, then, might consist solely of a movable hand or grasper that is able to pick up and move blocks. Many other types of robot problems have also been studied. In some problems the robot is a mobile vehicle that is to perform such tasks as moving objects from place to place through an environment cluttered with other objects.

Programming an actual robot involves integrating many functions, including perception of the robot's surrounding world, formulation of plans of action, and monitoring the execution of these plans. We will restrict our attention here to the problem of synthesizing a sequence of robot actions that (if properly executed) will achieve some stated goal, proceeding from some given initial situation.

The action synthesis part of the robot problem can be solved by a production system that has state descriptions corresponding to the various situations or "states of the world" in which the robot finds itself and whose F-rules correspond to the robot's actions.

B. State Descriptions and Goal Descriptions

We have already discussed how state descriptions and goals for robot problems can be constructed from predicate calculus wff's. For example, the robot "hand" and the configuration of blocks shown in Figure 8 can be represented by the conjunction of formulas shown in the figure. The "robot" in this situation is a simple "hand" that can move blocks about in a manner to be described below. The predicate HANDEMPTY has value T only when the robot hand is empty (as in the depicted situation.)

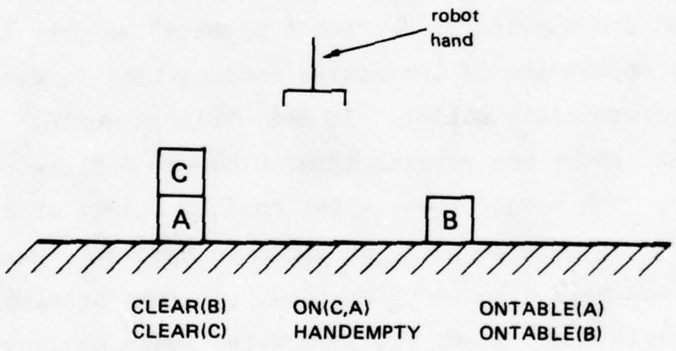

Figure 8 A Configuration of Blocks

Again, for the sake of simplicity, we will place certain restrictions on the kinds of formulas we will allow for describing world states and goals. For goal (and subgoal) expressions we will allow only simple conjunctions of literals. Any variables in goal expressions must be existentially quantified. For initial and intermediate state descriptions we will allow only conjunctions of ground literals (i.e. literals without variables). These restrictions are analogous to those imposed on the deduction systems described earlier. The formulas in Figure 8 clearly satisfy the restrictions.

C. Modeling Robot Actions

Robot actions transform one world state into another. We can model these actions by F-rules that change one state description into another. One simple but extremely useful technique for representing robot actions was proposed by Fikes and Nilsson [22] in a robot problem-solving system called STRIPS. The technique can be viewed as an elaboration of our use of implicational rules as F-rules. When we used an implication of the form W1 => W2 as an F-rule, W1 had to match expressions in the state description. Then the match instance of W2 was added to the state description. In particular, no terms were deleted from the state description. In modeling robot actions, however, F-rules must be able to delete expressions that might no longer be true. Suppose, for example, that the robot hand of Figure 8 were to pick up block B. Then the expression ONTABLE(B) would certainly no longer be true and should be deleted by any F-rule modeling this pickup action. STRIPS F-rules explicitly specify the expressions to be deleted by listing them.

A STRIPS F-rule consists of three components. The first is the precondition formula. This component is identical to the antecedent in an implicational F-rule. It is an expression that must be matched by expressions in the state description for the F-rule to be applicable. The second component is the delete list. It is a list of literals whose match instances are to be deleted from the old state description in constructing the new one. The third component is the add formula. Identical to the consequent of an implicational F-rule, its match instance is added to the state description.

For consistency with the restrictions on state description and goal wff's, we shall insist that the preconditions of our F-rules have existentially quantified variables only and that they consist of a conjunction of literals. Similarly, the add formula should also be limited to a conjunction of literals that contain no variables not included in the precondition formula. This latter restriction will ensure that any match instance of an add formula will be a conjunction of ground literals. (It is possible to lift some of these restrictions; we use them solely because they make our presentation much simpler.)

As an example, we shall describe the action of picking up a block from the table by a STRIPS-form F-rule. The preconditions for executing such an action are, say, that the block be on the table, that the hand be empty, and that the block have a clear top. The assumed result of the action is that the hand is holding the block and that all of the preconditions are deleted. We might represent such an action as follows:

```
pickup(x)
      Precondition:  ONTABLE(x) & HANDEMPTY & CLEAR(x)
      Delete list:  ONTABLE(x), HANDEMPTY, CLEAR(x)
      Add formula:  HOLDING(x)
```

Since, with our restrictions, the precondition and add formulas are conjunctions of literals, we can represent each of them by a set or list of literals. Sometimes, as in the above example, the precondition formula and the delete list will contain identical literals. In our example we have chosen to include only HOLDING(x) in the add formula rather than, additionally, the negations of literals in the delete list. For our purposes it will suffice merely to delete these literals from the state description.

We see that we could apply pickup(x) to the situation of Figure 8 only if B is substituted for x. The new state description in this case would be given by:

```
CLEAR(C)       ON(C,A)
ONTABLE(A)     HOLDING(B)
```

Because STRIPS-form F-rules may delete certain literals from a state description, production systems using them are not commutative. We are dealing here with F-rules that shift one set of states to another, as contrasted with F-rules based on implications, the application of which merely restricts the original set of states.

D. A Forward System

The simplest type of robot problem-solving system is a production system that uses the state description as the global data base and the F-rules as production rules. In such a system we select applicable F-rules to use until we produce a description that matches the goal expression. Let us examine how such a system might operate in a concrete instance. Consider, for example, the F-rules given below in STRIPS-form corresponding to a set of actions for the robot of Figure 8.

F-rules:

```
1)  pickup(x)
      P & D:  ONTABLE(x), CLEAR(x), HANDEMPTY
      A:  HOLDING(x)

2)  putdown(x)
      P & D:  HOLDING(x)
      A:  ONTABLE(x), CLEAR(x), HANDEMPTY

3)  stack(x,y)
      P & D::  HOLDING(x), CLEAR(y)
      A:  HANDEMPTY, ON(x,y), CLEAR(x)

4)  unstack(x,y)
      P & D:  HANDEMPTY, CLEAR(x), ON(x,y)
      A:  HOLDING(x), CLEAR(y)
```

(These F-rules are adapted from those in Dawson and Siklossy [23]). Note that in each of these rules the precondition formula (expressed as a list of literals) and the delete list are identical. The first rule is the same as the one we used as an example in the last section. The others are models of actions for putting down, stacking, and unstacking blocks.

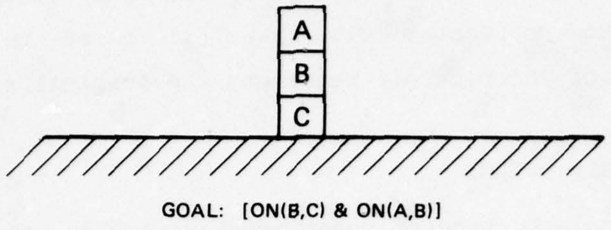

Figure 9 Goal for a Robot Problem

Suppose our goal is as shown in Figure 9. Working forward from the initial state description shown in Figure 8, we see that pickup(B) and unstack(C,A) are the only applicable F-rules. We show in Figure 10 the complete state space for this problem with a solution path indicated by the dark branches. The initial state description is labeled S0, and a state matching the goal is labeled G in Figure 10. (To reveal symmetries in the problem, we have not placed node S0 at the top in Figure 10.) Note that in this example each F-rule has an inverse.

In this very simple example (with only 22 states in the entire state space), a forward production system, even with an unsophisticated control strategy, can quickly find a path to a goal state. For more complex problems, however, we would expect that a forward search to the goal would generate a rather large tree, and that such a search would be feasible only if combined with a well-informed heuristic evaluation function.

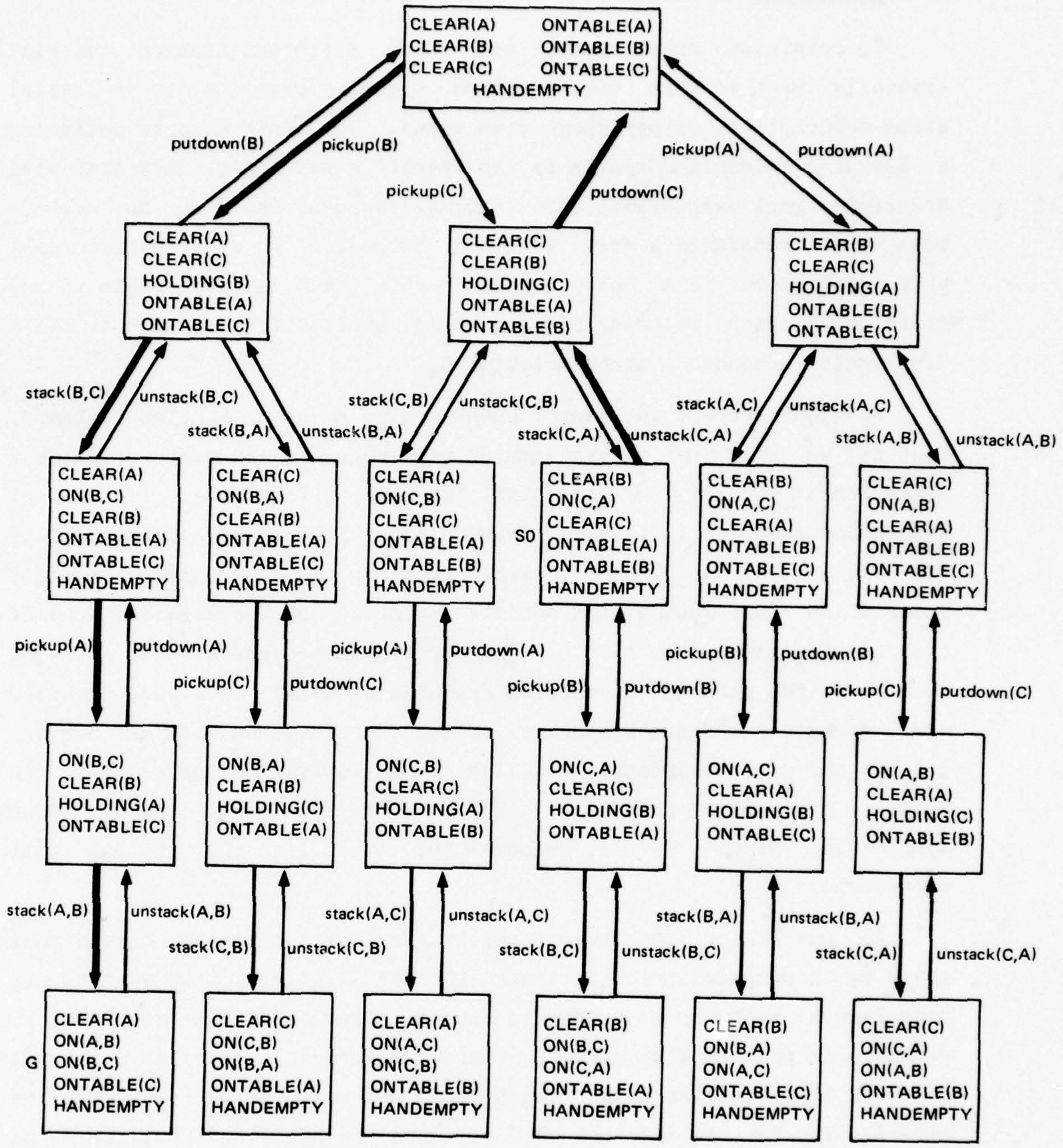
CLEAR(A) CLEAR(B) CLEAR(C) ONTABLE(A) ONTABLE(B) ONTABLE(C) HANDEMPTY
putdown(B)
pickup(B)
pickup(C)
putdown(C)
pickup(A)
putdown(A)
CLEAR(A) CLEAR(C) HOLDING(B) ONTABLE(A) ONTABLE(C)
CLEAR(C) CLEAR(B) HOLDING(C) ONTABLE(A) ONTABLE(B)
CLEAR(B) CLEAR(C) HOLDING(A) ONTABLE(B) ONTABLE(C)
stack(B,C)
unstack(B,C)
stack(B,A)
unstack(B,A)
stack(C,B)
unstack(C,B)
stack(C,A)
unstack(C,A)
stack(A,C)
unstack(A,C)
stack(A,B)
unstack(A,B)
CLEAR(A) ON(B,C) CLEAR(B) ONTABLE(A) ONTABLE(C) HANDEMPTY
CLEAR(C) ON(B,A) CLEAR(B) ONTABLE(A) ONTABLE(C) HANDEMPTY
CLEAR(A) ON(C,B) CLEAR(C) ONTABLE(A) ONTABLE(B) HANDEMPTY
S0
CLEAR(B) ON(C,A) CLEAR(C) ONTABLE(A) ONTABLE(B) HANDEMPTY
CLEAR(B) ON(A,C) CLEAR(A) ONTABLE(B) ONTABLE(C) HANDEMPTY
CLEAR(C) ON(A,B) CLEAR(A) ONTABLE(B) ONTABLE(C) HANDEMPTY
pickup(A)
putdown(A)
pickup(C)
putdown(C)
pickup(A)
putdown(A)
pickup(B)
putdown(B)
pickup(B)
putdown(B)
pickup(C)
putdown(C)
ON(B,C) CLEAR(B) HOLDING(A) ONTABLE(C)
ON(B,A) CLEAR(B) HOLDING(C) ONTABLE(A)
ON(C,B) CLEAR(C) HOLDING(A) ONTABLE(B)
ON(C,A) CLEAR(C) HOLDING(B) ONTABLE(A)
ON(A,C) CLEAR(A) HOLDING(B) ONTABLE(C)
ON(A,B) CLEAR(A) HOLDING(C) ONTABLE(B)
stack(A,B)
unstack(A,B)
stack(C,B)
unstack(C,B)
stack(A,C)
unstack(A,C)
stack(B,C)
unstack(B,C)
stack(B,A)
unstack(B,A)
stack(C,A)
unstack(C,A)
G
CLEAR(A) ON(A,B) ON(B,C) ONTABLE(C) HANDEMPTY
CLEAR(C) ON(C,B) ON(B,A) ONTABLE(A) HANDEMPTY
CLEAR(A) ON(A,C) ON(C,B) ONTABLE(B) HANDEMPTY
CLEAR(B) ON(B,C) ON(C,A) ONTABLE(A) HANDEMPTY
CLEAR(B) ON(B,A) ON(A,C) ONTABLE(C) HANDEMPTY
CLEAR(C) ON(C,A) ON(A,B) ONTABLE(B) HANDEMPTY

Figure 10 The State Space for a Robot Problem

E. Regression

To construct robot plans in a more efficient fashion we will typically want to work backward from a goal expression to an initial state description, rather than vice versa. Our first step in designing a backward production system is to specify a set of B-rules that will transform goal expressions into subgoal expressions. For any such B-rule that transforms a goal G into a subgoal G' we will want to make sure that there is a corresponding F-rule that is applicable to any state description matching G' and whose application to such a state description produces a state matching G.

We might attempt to specify such B-rules by using F-rules backward. That is, we know that application of an F-rule to any state produces a state that matches the add-list literals. Therefore, if a goal expression contains a literal L that unifies with one of the literals in the add list of an F-rule, we then know that, if we produce a state description that matches appropriate instances of the preconditions of that F-rule, the F-rule can then be applied to produce a state matching L. Thus, the subgoal expression produced by using an F-rule backward must certainly contain instances of the preconditions of that F-rule. But if the goal expression contains other literals (besides L), the subgoal expression must also contain other literals, which, after application of the F-rule, become the non-L literals in the goal expression.

To put all of this more formally, let us suppose we have a goal given by a conjunction of literals [L & G1 &...& GN]. We want to use some F-rule backward to produce a subgoal expression. Suppose that an F-rule with precondition formula P and add formula A contains a literal L' in A that unifies with L (with most general unifier u). Then the literals in Pu (the instance of P obtained by applying substitution u) are certainly a subset of the literals of the subgoal we seek. In addition, we must conjoin the literals G1',...,GN' with those in Pu to obtain a complete subgoal. The literals G1',...,GN' must be such that the application of the instance of the F-rule to any state description matching these literals will produce a state description matching G1,...,GN. Each Gi' is called the <u>regression</u> of Gi through the instance of the F-rule. The process of obtaining Gi' from Gi is called <u>regression</u>. (See Waldinger [24] for additional discussion of regression.)

For F-rules specified in the simple STRIPS-form, the regression procedure is quite easily described for ground instances of rules. Let R[Q;Fu] be the regression of a literal Q through a ground instance Fu of an F-rule with precondition, P, delete list, D, and add list, A. Then,

```
If Qu is a literal in Au
     R[Q;Fu] = T (True)

else, if Qu is a literal in Du

     R[Q;Fu] = F (False)

else, R[Q;Fu] = Qu
```

In simpler terms, Q regressed through an F-rule is trivially true if Q is one of the add literals, trivially false if Q is one of the deleted literals, or, if neither of the foregoing, it is Q itself.

F. <u>Interacting Goals</u>

We could use B-rules based on the F-rules of our sample problem above in a backward production system that starts with the goal expression [ON(A, B) & ON(B, C)] as its global data base. Such a system would ultimately produce a subgoal expression that matches the original state description. (The reader is invited to work out this example for himself. See Dawson and Siklossy [23] for their ideas on how such a system might work.)

When working backward on conjunctive goals (and their conjunctive subgoals), there is again a possibility of redundant multiple solutions that are equivalent, if different rule orderings are not considered. One way to avoid the redundancy of multiple solutions to the same goal component in different subgoals is to isolate a goal component and work on it exclusively until it is solved. After solving one of the components by finding an appropriate sequence of F-rules, we can return to the compound goal, select another component, and so on. This process is really tantamount to splitting or decomposing compound (i.e., conjunctive) goals into single-literal components and suggests the use of decomposable systems. Unfortunately, this simple strategy leads to severe difficulties when the F-rules are noncommutative. To illustrate these difficulties let us assume an attempt to split the compound goal of our sample problem, as depicted in Figure 11.

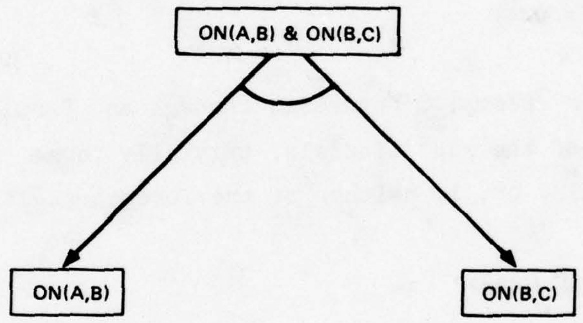

Figure 11 Splitting a Compound Goal

Suppose the initial state of the world is as shown in Figure 8. If we work on the component goal ON(B,C) first, we easily find the solution sequence {pickup(B), stack(B,C)}. But now the state of the world has changed, so that a solution to the other component goal ON(A,B) becomes more difficult. Furthermore, any solution to ON(A,B) from this state must "undo" the already achieved goal ON(B,C).

On the other hand, if we work on the goal ON(A,B) first, we find we can achieve it by the sequence {unstack(C,A), putdown(C), unstack(A,B)}. Again the state of the world has changed to one in which the other component goal, now ON(B,C), is harder to solve. There seems no way to solve this problem by selecting and solving one component, and then solving the other component without undoing the solution to the first.

We say that the component goals of this problem <u>interact</u>. Solving one destroys a solution to the other. In general, when the F-rules are noncommutative, as they are in this problem, a backward system cannot work on component goals separately. Interactions caused by the noncommutative effects of F-rule applications prevent us from being able to use successfully the strategy of combining separate solutions for each component. Thus we cannot develop a backward, decomposable system based on these STRIPS rules.

In our sample problem the component goals are highly interacting. More typically, we would expect only moderate interactions. In such cases it might be more efficient to assume initially that compound goals can be split -- and to handle interactions, when they arise, by special mechanisms, rather than assume that all compound goals are likely to interact. In the next section we shall describe a problem-solving system named STRIPS based on this general strategy.

G. <u>STRIPS</u>

The STRIPS system was one of the early robot problem-solving systems [22]. STRIPS maintains a "stack" of goals and focuses its problem-solving effort on the top goal of the stack. Initially the goal stack contains just the main goal. Whenever the top goal in the stack matches the current state description, it is eliminated. Otherwise, if the top goal in the stack is a compound one, STRIPS decomposes it and adds each of the component goal literals in some order above the compound goal in the goal stack. The idea is that STRIPS will work on each of these component goals in the order in which they appear on the stack. When all of the component goals are solved, the system will reconsider the compound goal again, relisting the components at the top of the stack if the compound goal does not match the current state description. This reconsideration of the compound goal is the safety feature that STRIPS uses to deal with the interacting-goal problem. If solving one component goal undoes an already solved component, the undone goal will be reconsidered and resolved if needed.

When the top (unsolved) goal in the stack is a single-literal goal, STRIPS looks for an F-rule whose add list contains a literal that can be matched to that top goal. The match instance of this F-rule then replaces the single-literal goal at the top of the stack. Above the F-rule is then added the match instance of its precondition formula, P. If P is compound and does not match the current state description, its components are added above it, in some specified order, to the stack.

If the top item on the stack is an F-rule, this is so because the precondition formula of the F-rule was just removed from the stack when its precondition formula was matched by the current state description. Since the F-rule is therefore applicable, it is applied to the current state description and removed from the top of the stack. The new state description now replaces the original one, and the system keeps track of the F-rule that has just been applied for later use in composing a solution sequence. We can view STRIPS as a production system in which the global data base combines the current state description and the goal stack. Operations on this data base result in changes to either the state description or the goal stack, a process that continues until the goal stack is empty. Thus, the "rules" of this production system are those that transform one global data base into another. They should not be confused with the F-rules that correspond to the models of robot actions. (Here we have an example of a production system within a production system. Top-level rules change the global data base consisting of both state description and goal stack. The F-rules are named in the goal stack and are used to change the state description.)

The operation of the STRIPS system with a tree-search control regime produces a tree of global data bases; a solution corresponds to a path in this tree leading from the start to a termination node. (A termination node is one with an empty goal stack.)

Let us see how STRIPS might solve a simple block-stacking problem. Suppose the goal is [ON(C,B) and ON(A,C)], and the initial state is as shown in Figure 8. We note that this goal can be accomplished simply by putting C on B, then A on C. We will use the same F-rules as before.

In Figure 12 we show part of a tree that might be generated by STRIPS during the solution of this sample problem. Since this problem was very simple, STRIPS quite easily produces the solution sequence {unstack(C,A), unstack(C,B), pickup(A), stack(A,C)}.

STRIPS has somewhat more difficulty with the problem whose goal is [ON(B,C) & ON(A,B)]. Starting from the same initial configuration of blocks, it is possible for STRIPS to produce a solution sequence longer than needed, namely, {unstack(C,A), putdown(C), pickup(A), stack(A,B), unstack(A,B), putdown(A), pickup(B), stack(B,C), pickup(A), stack(A,B)}. The third-through-sixth F-rules represent an unnecessary detour. In this case, the detour results because STRIPS decided to achieve ON(A,B) before achieving ON(B,C). The interaction between these goals then forced STRIPS to undo ON(A,B) before it could achieve ON(B,C).

H. Control Strategies for STRIPS

Several decisions must be made by the control component of the STRIPS system. We'll mention some of these briefly. First, when a compound goal is split into individual components we must order these components on the goal stack. A reasonable approach is to isolate all components that match the current state description. (Conceptually they can be put on the top of the stack and then immediately stripped off.) This leaves only the unmatched goals to be ordered. We could create a new successor node for each possible ordering (as we did in our examples), or we could select just one of them arbitrarily (perhaps that goal literal heuristically judged to be the most difficult) and create a successor node in which only that component goal is put on the stack. This latter approach is probably adequate because, after this single

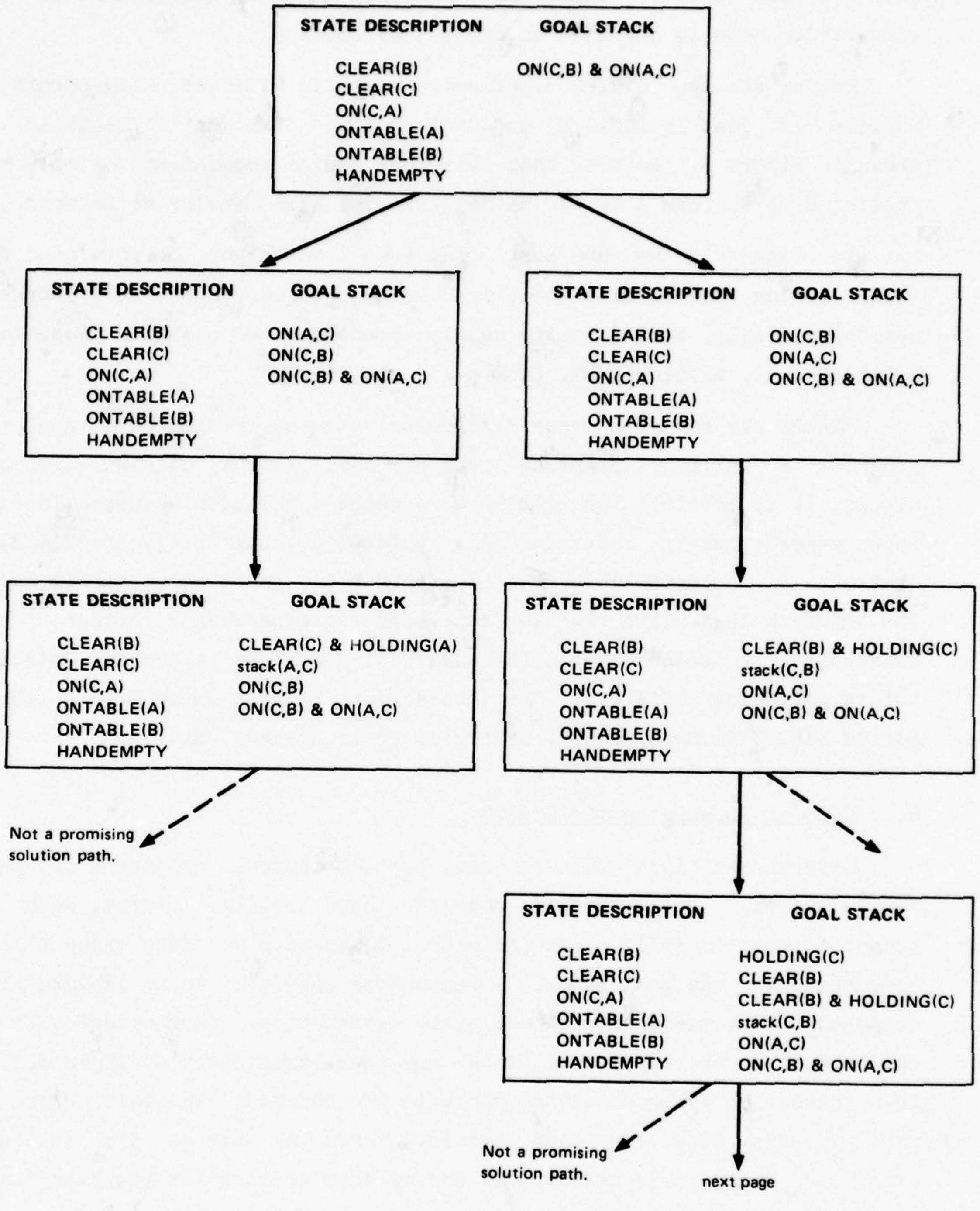

Figure 12 A Problem-Solving Tree Produced by STRIPS

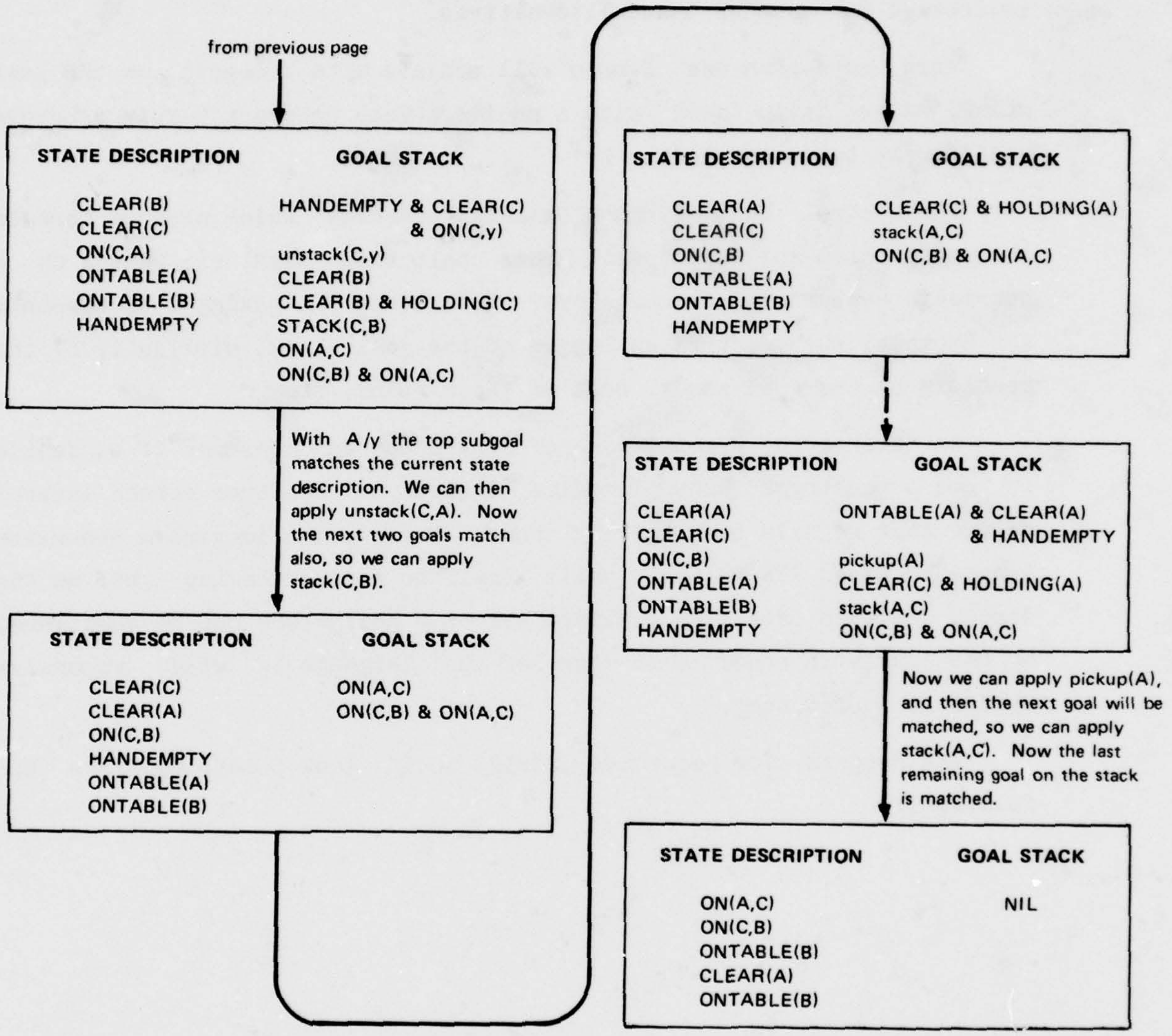

Figure 12 A Problem-Solving Tree Produced by STRIPS (Concluded)

goal is solved, confronting us again will be the compound goal and the opportunity to select another of its unachieved components.

When (existentially quantified) variables occur in the goal stack, the control component may need to make a choice from among several possible instantiations. We can assume that a different successor node is created for each of these alternatives.

When more than one F-rule will achieve the top goal on the goal stack, we are again faced with a choice. Each relevant F-rule produces a different successor node.

The control strategy must be able to decide which node to work on in the problem-solving tree. This selection might be based on a heuristic evaluation function over these nodes -- taking into account, for example, such factors as length of the goal stack, difficulty of the problems on the goal stack, cost of the F-rules, etc.

An interesting special case of STRIPS can be developed if we decide to use a backtrack control regime instead of the tree search control regime that we have been talking about. Here we can imagine a recursive function named STRIPS that calls itself to solve the top goal on the stack. In this case, the explicit use of a goal stack can be supplanted by the built-in stack mechanism of the language in which recursive STRIPS is implemented.

The program for recursive STRIPS would look something like the following:

First we set S, a global variable, to the initial state description. (We call the program originally with the argument G, the goal that STRIPS is trying to achieve.)

```
STRIPS (G)
1.  Begin
2.     Until S matches G, do:; the main loop of STRIPS is iterative
3.        begin
4.          g <- a component of G that does not match S;  a non-
                 deterministic selection and therefore a backtracking
                 point
5.          f <- an F-rule whose add list contains a literal that
                 matches g; another backtracking point
6.          p <- precondition formula of appropriate instance of f.
7.          STRIPS(p); a recursive call to solve the subproblem
8.          S <- result of applying appropriate instance of f to S
9.        end
10. End
```

This version of STRIPS is analogous to the recursive version of GPS developed by Newell, Shaw, and Simon [25]. Referring to the above algorithm, "g" is analogous to the GPS "difference," "f" is analogous to the operator that reduces the difference, and "p" is the subgoal that must first be achieved in order to apply the operator.

Several other schemes have been developed that attempt to decompose compound goals, solve them by a backward production system, and then modify the solution to deal with interactions. Waldinger [24] and Reiger and London [26] describe methods that involve finding a sequence of F-rules for solving one conjunct of a goal and then adding other F-rules appropriately to the sequence so that the modified sequence solves the entire conjunction. Sussman [27], Sacerdoti [28], and Tate [29] describe techniques that initially ignore possible interactions to develop a trial solution to a conjunctive goal. This solution is then "debugged" or "criticized" to eliminate interactions. Sacerdoti's system also introduced the idea of a "nonlinear" plan, in which the ordering of the F-rules was left partial until forced otherwise by interaction-eliminating critics. Another important idea for producing sequences of F-rules efficiently involves the use of hierarchies of "abstraction spaces." Sacerdoti's ABSTRIPS system [30] works through a series of hierarchical levels, gradually considering more and more detailed sets of F-rule preconditions. At each level problem-solving activity improves upon an approximately adequate plan produced by the prior level and passes this plan on down for further improvement. Sacerdoti's NOAH system [28] combined similar hierarchical strategies with his previously mentioned ideas for dealing with interaction problems.

As an example of the fact that the same problem can be solved by totally different types of systems, Kowalski [15] and Green [31] developed techniques for solving robot problems with theorem-proving-type systems using commutative F-rules. Kowalski's approach can be viewed as a way of implementing STRIPS-type problem solvers in a PROLOG-like language.

This completes our general description of problem-solving systems. All of these systems used a rather simple version of a predicate calculus language. Next we shall describe certain important elaborations of this language.

VI Semantic Network Representations

A. Motivation

Various representational formalisms (and informalisms) have been suggested as alternatives to the predicate calculus in AI systems. Examples are frame systems [32], SCRIPTS [33], unit systems [34], KRL [35], FRL [36], and semantic networks [6,37]. Upon close analysis most of these systems can be regarded largely as syntactic variants of predicate calculus in which various indexing, taxonomic, and symbol-typing conventions are included within the formalism. A feature of these alternative representations is that certain of the operations that might otherwise have to be performed by explicit rule applications are performed instead in a more automatic way by mechanisms that depend on the structure of the representation. The phrase structured objects is often used to describe some of these representational schemes because of the heavy emphasis on the structure of the representation; indeed, the structure itself carries some of the representational and computational burden. In this section we shall describe and provide a rationale for one type of structured object representation.

Most notations for structured objects involve the use of binary (two-argument) predicates for expressing facts about the objects. A simple conversion scheme can be used to re-express arbitrary wff's using only binary predicates. (See Deliyanni and Kowalski [38] for more discussion about this scheme.) For example, to convert the formula GIVE(JOHN, MARY, BOOK) (representing "John gave Mary the book,") we first postulate a set of giving events, denoted by GIVING-EVENTS. We then assert the existence of a member of this set, and relate it by new binary predicates to each argument of the original predicate. Using this scheme, the formula GIVE(JOHN, MARY, BOOK) would be converted to:

```
(EXISTS x){EL(x, GIVING-EVENTS) & GIVER(x, JOHN)
           & RECIP(x, MARY) & OBJ(x, BOOK)}
```

The predicate EL is used to express set membership. Skolemizing the existential variable in the above formula gives a name, say G1, to our particular giving event:

```
EL(G1, GIVING-EVENTS) & GIVER(G1, JOHN)
 & RECIP(G1, MARY) & OBJ(G1, BOOK)
```

Thus, we have converted a three-argument predicate to the conjunction of four binary ones.

The relations between G1 and the original arguments of GIVE might just as well be expressed by functions over the set GIVING-EVENTS, rather than by predicates. With this additional notational change, the sentence "John gave Mary the book" can be represented by the following formula:

```
EL(G1, GIVING-EVENTS)
   & EQ[giver(G1), JOHN]
   & EQ[recip(G1), MARY]
   & EQ[obj(G1), BOOK]
```

The predicate EQ is meant to denote the equality relation. The expression above uses certain functions, defined over the set GIVING-EVENTS, whose values name other objects that participate in G1.

There are some advantages in converting to a representation that uses events and binary relations. For our purposes the primary benefit is modularity. Suppose, for example, that we wanted to add information about the time at which a giving event took place. Before converting to our binary form, we would have had to add a fourth (time) argument to the predicate GIVE. Such a change might require extensive modifications of the production rules that refer to GIVE, as well as of the control system. If the notion of giving is represented instead as a domain entity, additional information about it can easily be incorporated by adding new binary relations.

In this section we shall represent all but a small number of propositions as "events" or "situations" that are entities of our domain. The only predicates we will need are EQ (to say that two entitities are the same), SS (to say that one set is a subset of another), and EL (to say that an entity is an element of a set).

B. A Graphical Representation

The binary-predicate version of predicate calculus introduced in the last section lends itself to graphical representation. The terms of the formalism (namely, the constant and variable symbols and the functional expressions) can be represented by nodes of a graph. Thus, in our examples above, we would have nodes for JOHN, G1, MARY, and BOOK. The predicates EQ, EL and SS can be represented by arcs; the tail of the arc exits from the node representing the first argument, while its head enters the node representing the second argument. Thus, the expression EL(G1, GIVING-EVENTS) is represented by the following structure:

```
 0-------------------------->0
G1           EL             GIVING-EVENTS
```

The nodes and arcs of such graphs are labeled by the terms and predicates they denote.

When an EQ predicate relates a term with a unary function of another term, we shall represent the unary function expression by an arc connecting the two terms. For example, to represent the formula EQ[giver(G1), JOHN], we use the structure:

```
 0--------------------------->0
G1          giver            JOHN
```

A collection of predicate calculus expressions of the type we have been discussing can then be represented by a graph structure that is often called a <u>semantic network</u>. A network representation of our sample collection of sentences is shown in Figure 13. Semantic networks of this sort are useful for descriptive purposes because they give a simple, structural picture of a body of facts. They also depict some of the indexing structure used in implementations of representations. Of course, whether we choose to describe the computer representation of a certain body of facts as a semantic network or as a collection of linear formulas is mainly a matter of taste. The underlying computer data structures may well be identical! We shall use both types of descriptions more or less interchangeably in this section.

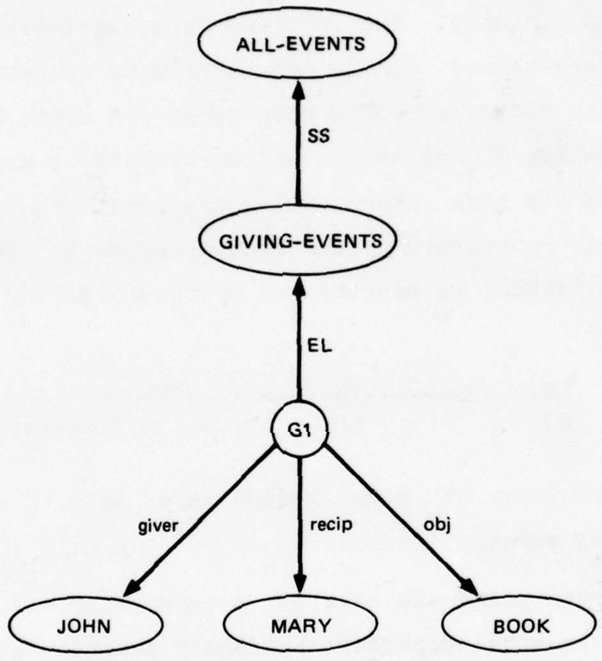

Figure 13 A Simple Semantic Network

The nodes in the networks of Figure 13 are all labeled by constant symbols. We will also allow variable nodes. These will be labeled by lowercase letters from the end of the alphabet (e. g. ...,x, y, z). The variables are standardized apart and are assumed to be universally quantified. The scope of these quantifications encompasses the entire fact network. Existentially quantified variables are Skolemized, and the resulting Skolem functions are represented by nodes that are labeled by functional expressions. Thus, the sentence "John gave something to everyone" can be represented by the network in Figure 14. In this figure "x" is universally quantified. The nodes labeled by "g(x)" and "sk(x)" are Skolem-function nodes. (Computer implementations of nodes labeled by functional expressions would probably have some sort of pointer structure between the dependent nodes and the independent ones. For simplicity we shall suppress explicit display of these pointers in our semantic networks, although some net formalisms do include them [39].)

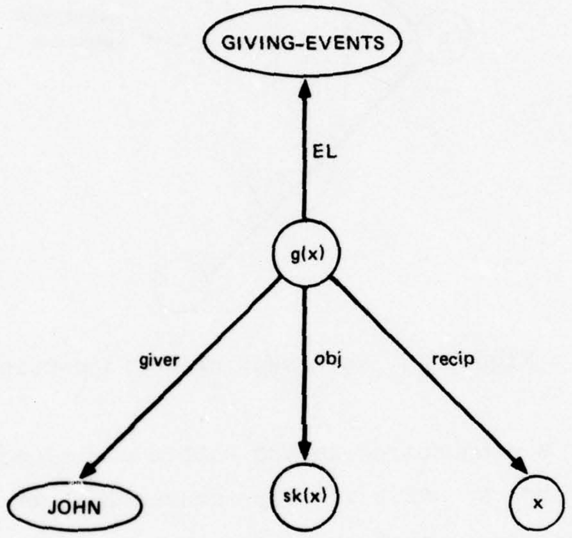

Figure 14 A Net with Skolem-Function Nodes

We next discuss how we are going to represent the propositional connectives graphically. Representing conjunctions is easy; the multiple nodes and EL and SS arcs in a semantic network represent the conjunction of the associated atomic formulas. To represent a disjunction we need some way of setting off those nodes and arcs that are the disjuncts. In a linear notation we use parentheses to delimit the disjunction. For semantic networks we shall employ a graphical version of parentheses, an <u>enclosure</u>, represented by a closed, dashed boundary. For a disjunction each disjunctive predicate is drawn within the enclosure, and the enclosure is labeled "DIS." Thus, the expression [EL(A, B) v SS(B, C)] is represented as in Figure 15.

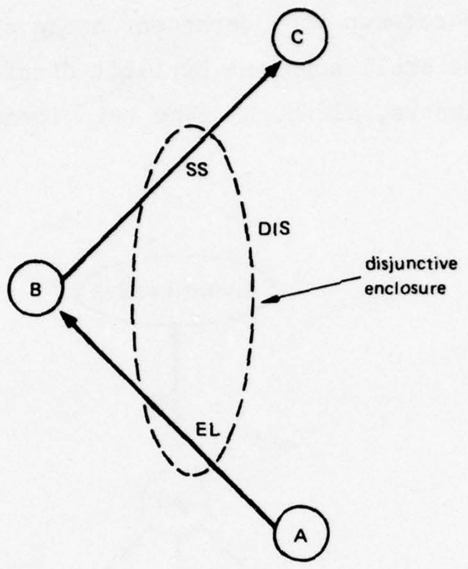

Figure 15 Representing a Disjunction

To set off a conjunction nested within a disjunction, we can use an enclosure labeled by "CONJ." (By convention we omit the implied conjunctive enclosure that surrounds the entire semantic network.) Arbitrary nesting of enclosures within enclosures can be handled in this manner. As an example, we show in Figure 16 the semantic-network version of the sentence "John is a programmer or Mary is a lawyer."

We can use enclosures to delimit the scope of negations as well. In this case, we label the enclosure "NEG." We show in Figure 17 a graphical representation of ~[EL(A, B) & SS(B, C)]. To simplify the notation we assume by convention that the predicates within a negative enclosure are conjunctive. In Figure 18 we show an example of a semantic network with both a disjunctive and a negative enclosure. This network is a representation for the statement "John bought a Ford or a Chevy. It was not a convertible."

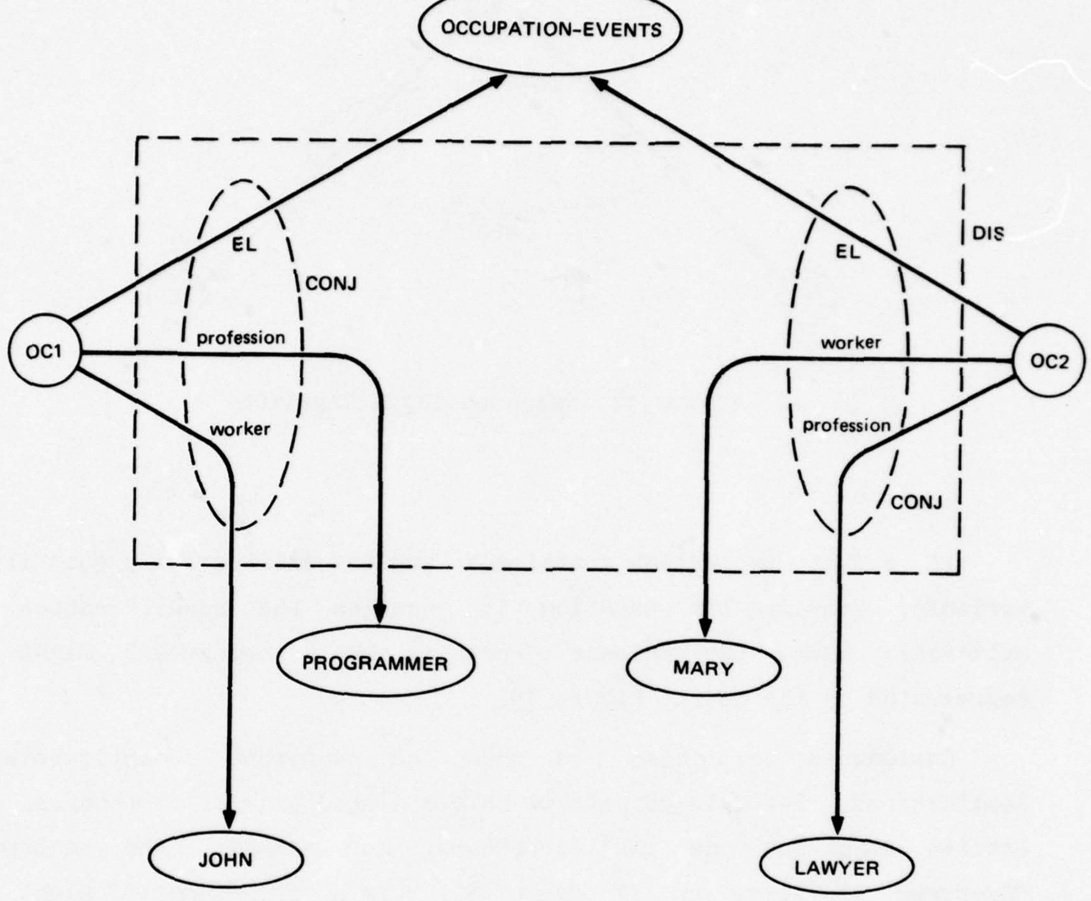

Figure 16 A Disjunction with Nested Conjunctions

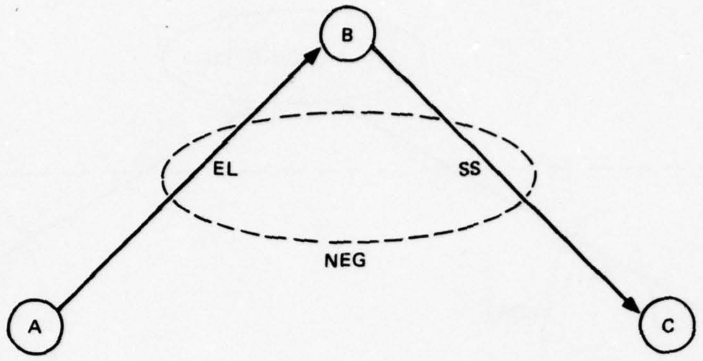

Figure 17 Representing a Negation

If a fact expression contains a negated existentially quantified variable, moving the negation in changes the quantification to universal. Thus, the sentence "Mary is not a programmer" might be represented by the net in Figure 19.

Enclosures can also be used to represent semantic-network implications. For this purpose we have a linked pair of enclosures, one labeled ANTE and one labeled CONSE. For example, the sentence, "Everyone who lives at 37 Maple St. is a programmer," might be represented by the net in Figure 20. In this figure o(x,y) is a Skolem function naming an occupation event dependent on x and y. A dashed line links the ANTE and CONSE enclosures to show that they belong to the same implication. We shall discuss network implications in more detail later.

In all these examples enclosures are used to set off a group of EL, SS, and function arcs, and thus are drawn so as to enclose only arcs.

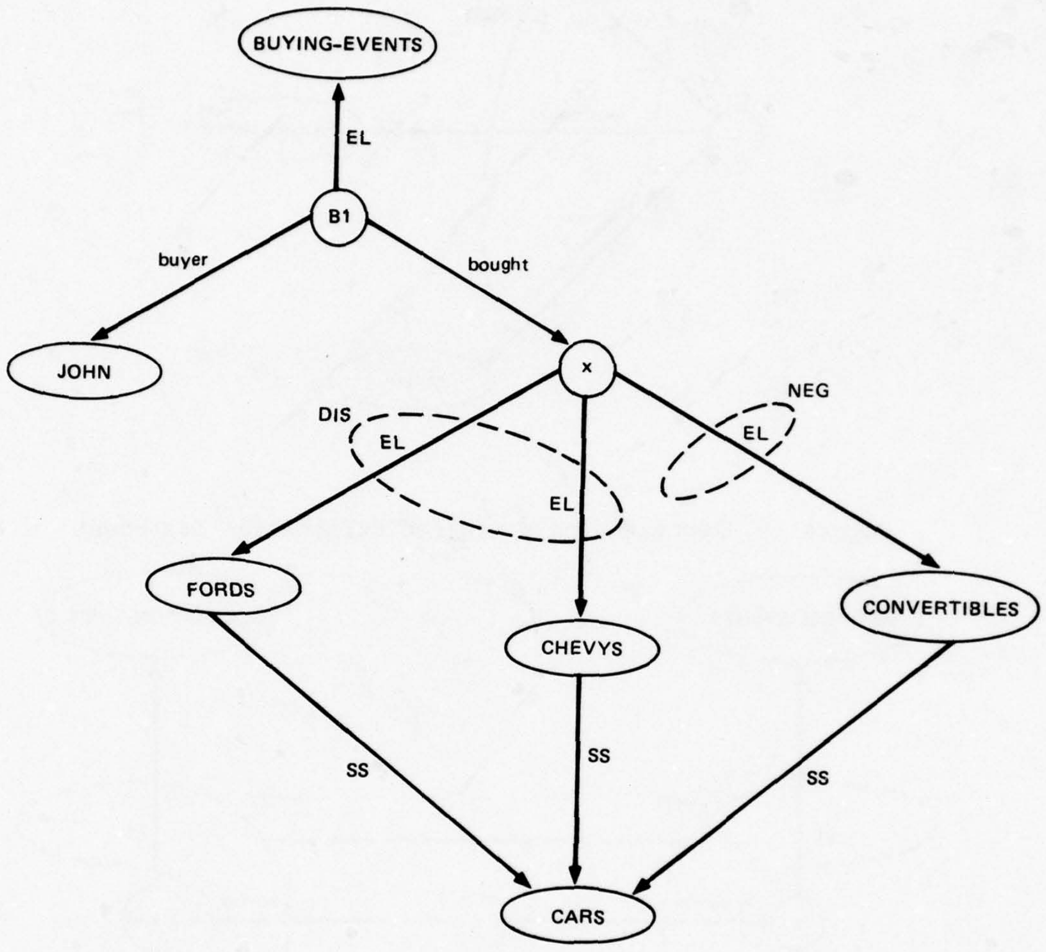

Figure 18 A Semantic Network with Logical Connectives

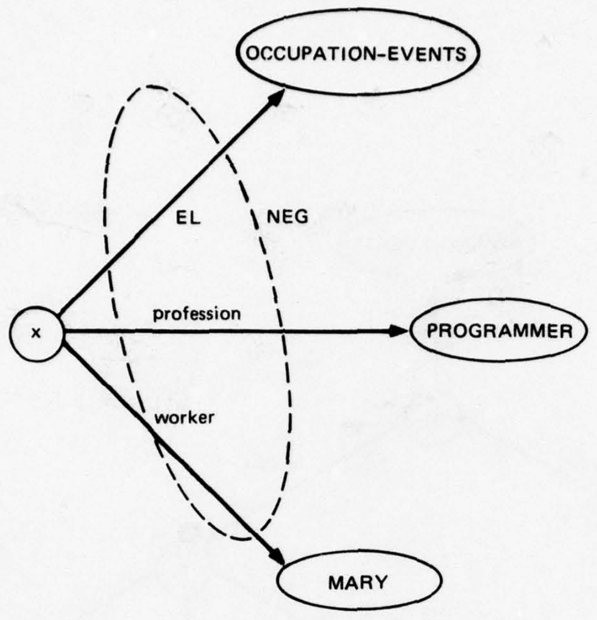

Figure 19 Representing a Negated Existential Statement

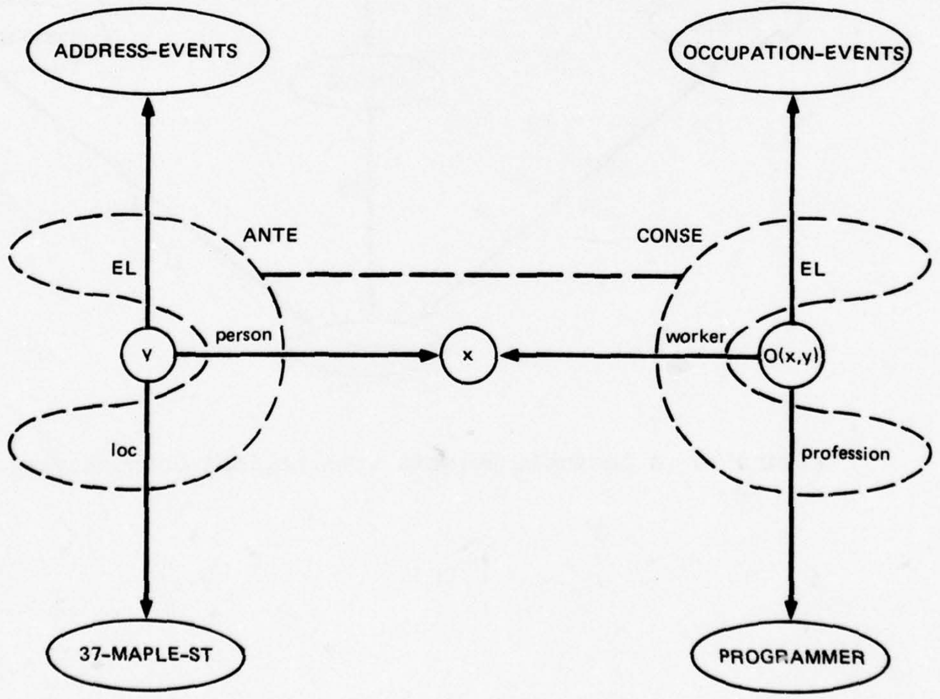

Figure 20 A Network with an Implication

(Whether or not they enclose nodes is of no consequence in our semantic-net notation.)

C. Matching

To use semantic networks as the global data base of a production system, we need to discuss the nature of the production rules that are employed in altering the data base. A critical element in using such rules is a matching operation that is analogous to unification. We turn to this subject next.

To help us decide what we mean by asking whether two semantic networks "match," we must depend on the fact that semantic networks are an alternative kind of predicate calculus formalism. The appropriate interpretation must involve a definition such as this: two networks match if and only if the predicate calculus formula associated with one of them unifies with the predicate calculus formula associated with the other. We shall be interested in a somewhat weaker definition of match, because our match operations will usually not be symmetrical. That is, we shall usually have a goal network that we want to match with a fact network. We shall say that a goal network matches a fact network if the formula involving the goal network unifies with some subconjunction of the formulas of the fact network. (Matching could then occur only if the goal network formulas were provable from the fact network formulas.)

In Figure 21 and Figure 22 we show examples of fact and goal networks. In these figures we separate the fact and goal structures by a dashed line and allow them to share nodes. (In semantic networks, when two terms are identical they are always represented by the same node.) In these examples we merely have to find fact arcs that match each of the goal arcs. The match is successful in Figure 21, but not in Figure 22.

In any representational scheme there are often several alternatives for representing basically the same information. Since our definition of structure matching depends on the exact form of the structure, such alternatives do not strictly match one another. Consider the network

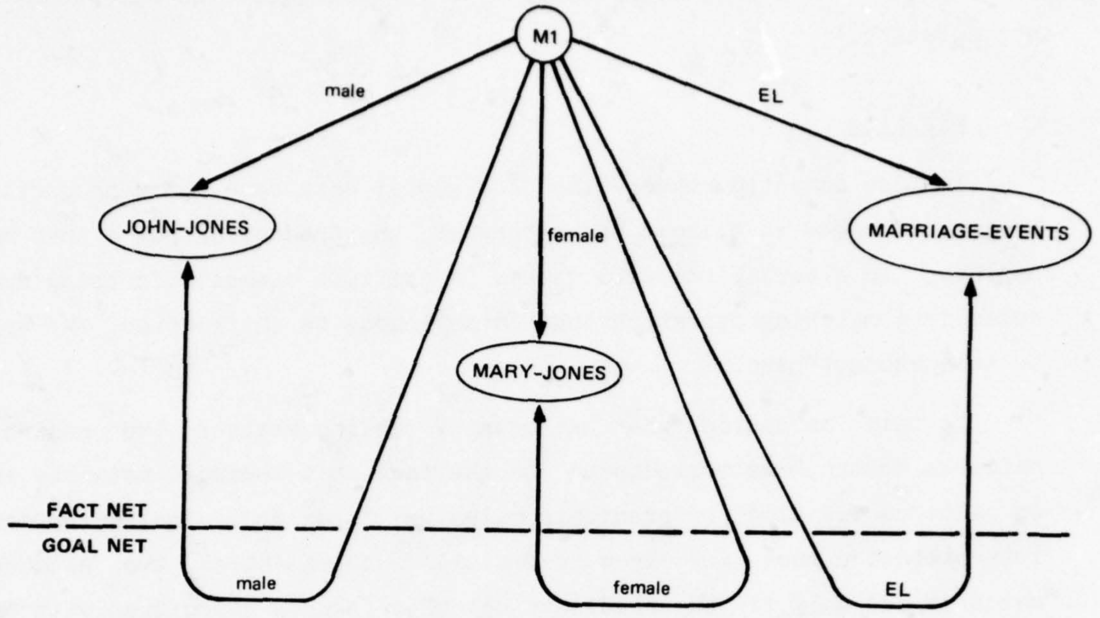

Figure 21 A Goal Net that Matches a Fact Net

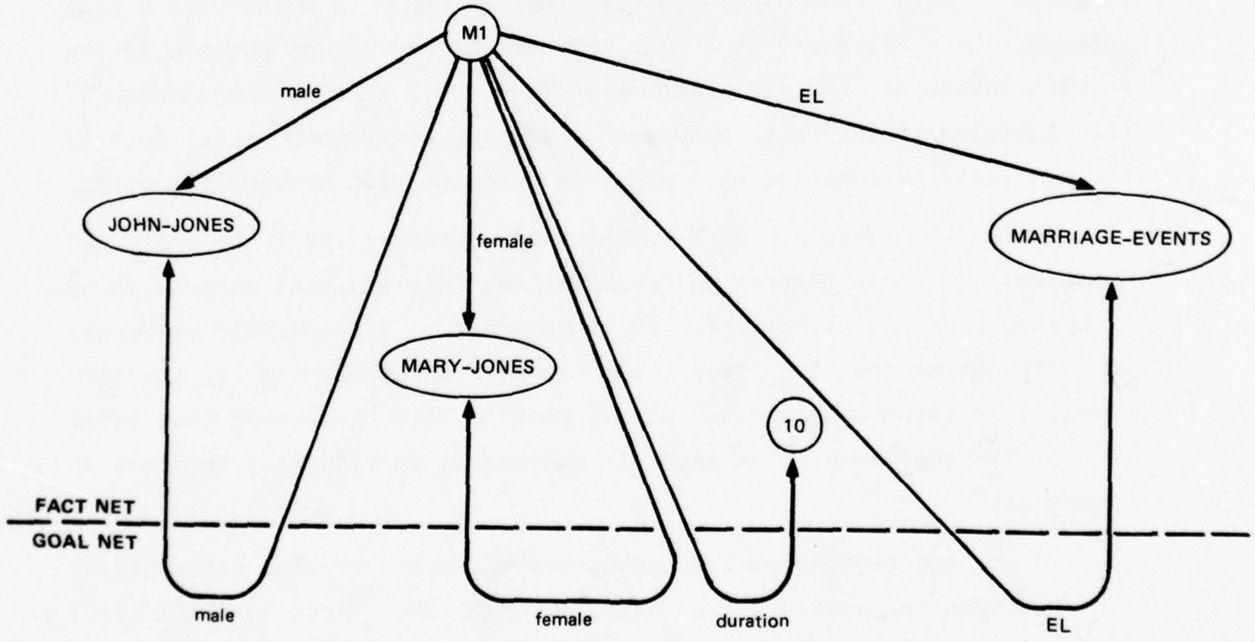

Figure 22 A Goal Net that Does Not Match a Fact Net

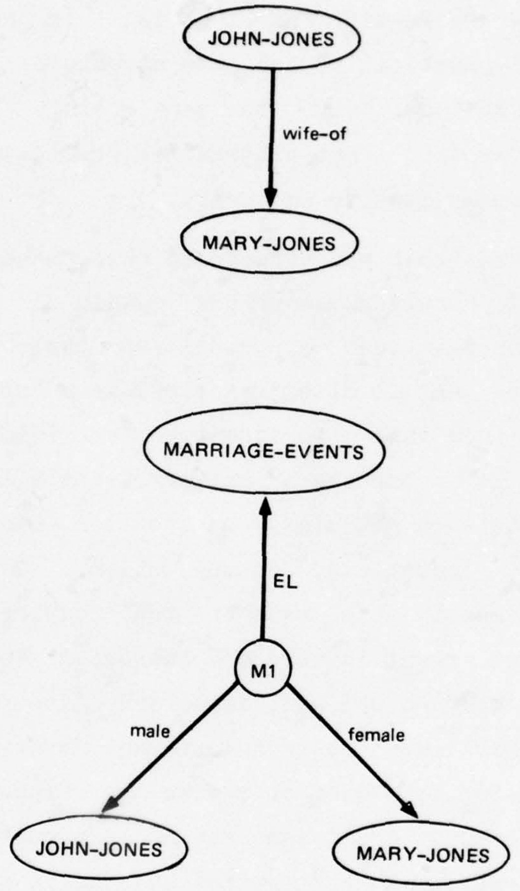

Figure 23 Two Nonmatching Yet Equivalent Structures

examples of Figure 23. There we show two alternatives for representing "John Jones is married to Mary Jones." One of these uses a "marriage-event," and the other uses the special "wife-of" function. (Ordinarily, our preference is not to use functions like "wife-of," unless their values are truly independent of other parameters, such as time. In particular, the "wife-of" function is, in real life, sometimes impermanent.) Syntactically, the two structures of Figure 23 do not match, even though they "say" the same thing. Such a circumstance corresponds to the fact that alternative predicate calculus forms for representing the same idea do not unify.

Some AI systems that use structured objects have elaborate matchers that use special knowledge about the domain of application to enable direct matches between such structures as those shown in Figure 23. These systems have what is often described as a "semantic matcher," that is, one that decides that two structures are identical if they "mean" the same thing. It is perhaps a matter of taste as to where we want to draw the line between matching, on the one hand, and more complex computations and deductions, on the other. Our preference is to prohibit operations in the matcher that require specialized domain knowledge or that might involve combinatorial computations. In these cases we would prefer to use rule-based deductive machinery to establish the semantic equivalence between different syntactic forms. Such a strategy retains for the control system the responsibility for managing all potentially combinatorial searches. It permits the matcher to be a general-purpose routine that does not have to be specially designed for each application. We shall be talking about deductive machinery later when we discuss operations on structured objects.

A common cause of syntactic differences between network structures arises because of different ways of setting up chains of EL and SS arcs. Consider the example of Figure 24. The goal structure can be derived from the fact structure by means of a fundamental theorem from set theory. Because this derivation occurs so often with structured objects, it is usually built into the matcher. In fact, one of the

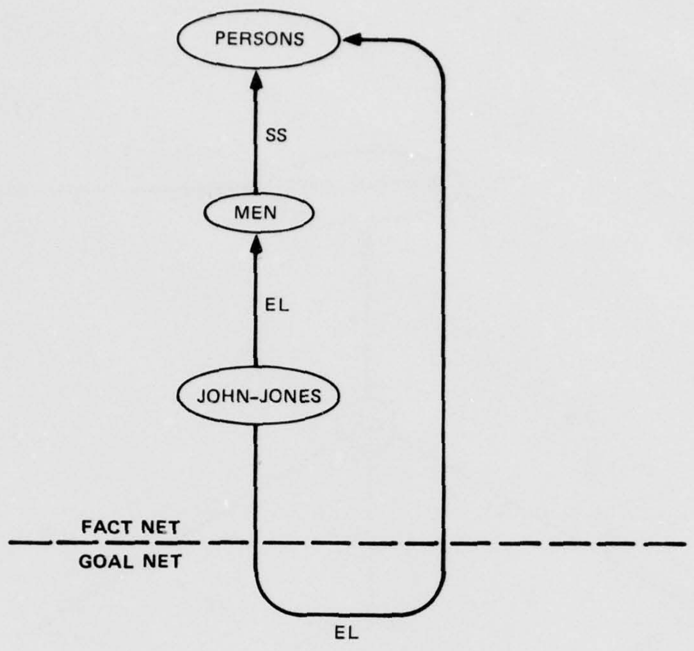

Figure 24 Nets with EL and SS Arcs

advantages of structured objects is that their pointer structures allow easy computation of element/subset/set relationships. Thus, we shall say that the two structures in Figure 24 do indeed match.

So far, we have discussed matching only between two constant structures. Usually one or both of the structures will contain variables that can have terms substituted for them during the matching process. Variables that occur in fact structures will have implicit universal quantification in all formulas in which they appear, while variables that occur in goal structures will have implicit existential quantification in all formulas in which they appear.

In Figure 25 we show an example of a fact net that matches a goal network containing a variable. For a goal net to be matched each of its elements (arcs and nodes) must unify with corresponding fact net elements.

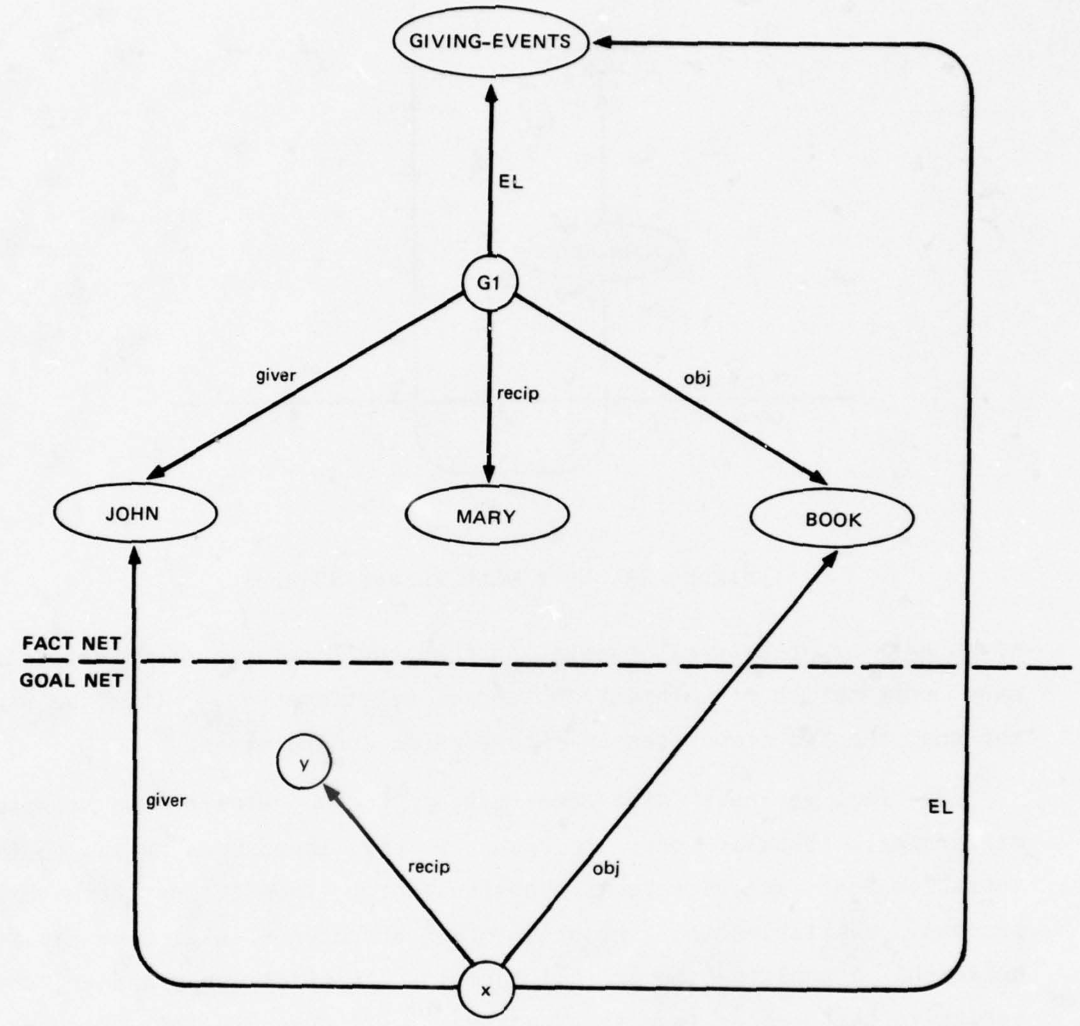

Figure 25 Matching Nets

D. Operations on Structured Objects

1. Delineations

Semantic network representations can be used in production systems for performing deductions. Just as in our earlier discussion of predicate calculus deduction systems, the production rules can be based on implications. Before talking about how we will use implications in general, we first consider a frequently occurring special use in which an implication is used to assert properties about every member of a given set. Consider, for example, the sentence "All computer science students have graduate standing." From it and the sentence, "John is a computer science student," we should be able to deduce that "John has graduate standing." We could represent this syllogism in the predicate calculus as follows:

Fact: EL(JOHN, CS-STUDENTS)

Rule: EL(x, CS-STUDENTS) => EQ[class(x), GRAD]

Goal: EQ[class(JOHN), GRAD]

An ordinary predicate calculus production system could use the rule, in either direction, to establish the goal.

When we use networks, our fact and goal expressions might be represented as shown in Figure 26. We could, of course, represent the implicational rule by a network implication structure, but in this case we prefer a simpler construction called a <u>delineation</u>. Such a construction describes (delineates) each of the individuals in a set. For example, the delineation describing each individual in the set of computer science students is shown in the network of Figure 27. It consists of a node labeled by a "typed" universal variable whose domain of universal quantification is the set denoted by CS-STUDENTS. The type of the variable, that is, the name of its domain set, follows the variable after a vertical bar, "|."

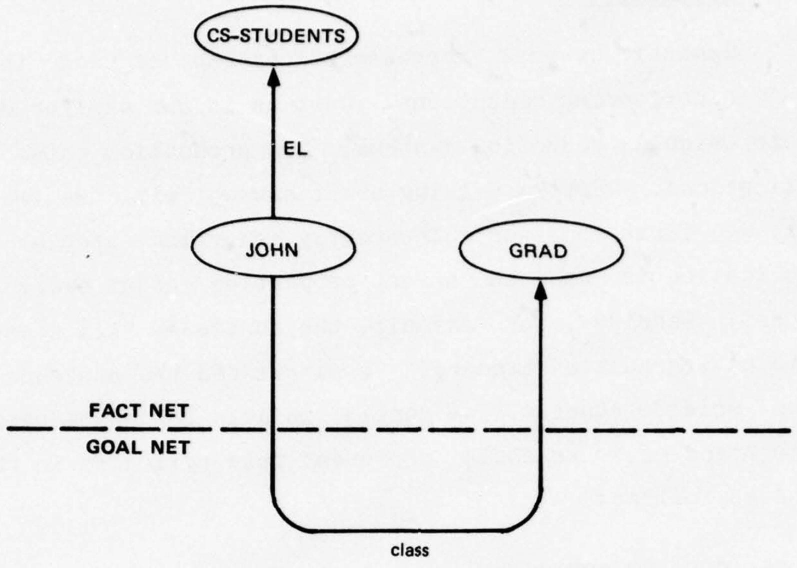

Figure 26 Fact and Goal Nets

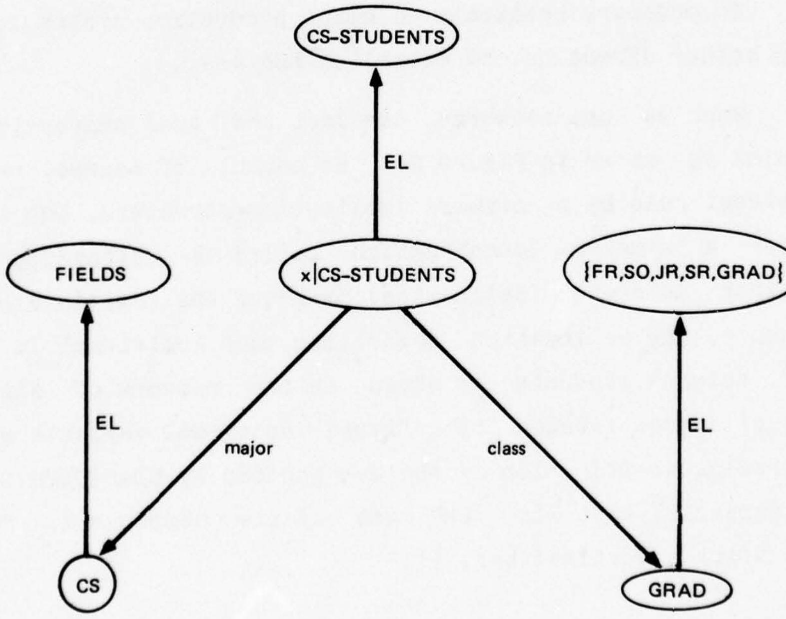

Figure 27 A Network Delineation for CS-STUDENTS

Delineations can be used in the forward direction to create new network structure. For example, consider the fact network shown in Figure 26. To use the delineation in the forward direction, we note that the node x¦CS-STUDENTS matches the fact node JOHN. The typed variable "x" matches any term that is an element of CS-STUDENTS. Applying the delineation to the fact network involves adding an arc "major" pointing to a node "CS" and an arc "class" pointing to a node "GRAD." Thus extended, the new fact network matches the goal net shown in Figure 26.

Used in the backward direction on this same goal network, the delineation establishes the subgoal of showing that John is an element of the set of computer science students. The network for this subgoal matches the original fact network, so we again have a proof.

We must be careful not to confuse delineations describing each individual in a set with the node describing the set itself or with nodes describing particular individuals in the set! Some AI systems using these kinds of formalisms have entities called prototypes that seem to play the same role as do our delineations. In these systems prototypes seem to be treated as if they described a mythical "typical" member of a set. The prototypes are then related to other members of the set by an "instance" relation. But there is potential for confusion here, because substituting a constant for a variable should properly be thought of as a metaprocess, rather than as a function in the formalism itself. It seems more reasonable to think of a delineation as a special form of implicational rule.

E. Property Inheritance

In many applications the network denoting individuals and sets forms a taxonomic hierarchy. The tree-like taxonomic division of animals into species, families, orders, etc., is one example. The taxonomies used in ordinary reasoning might be somewhat more "tangled" than those used in formal biology -- an individual may be an element of more than one set, for example. Usually, though, useful hierarchies converge toward a small number of sets at the top, and, in any case, the various sets form a partial order under the subset relation.

Consider the sample hierarchy shown in Figure 28. Upon learning that Clyde, say, is an elephant, we could use the delineations (together with some set theory) to draw several forward inferences. Specifically, we could derive the information that Clyde is gray and wrinkled, that he likes peanuts, that he is warm-blooded, etc. The results of these operations could be used to augment the network denoting Clyde. In any given reasoning problem, we obviously would not want to derive all of these facts about Clyde explicity. Uncontrolled reasoning (as usual) is quite inefficient.

Similar efficiency problems arise in using delineations in a taxonomic hierarchy to reason backward. Suppose, say, that we wanted to prove that Clyde was gray when this fact was not reresented explicitly. Using the delineations of Figure 28, we might set up several subgoals including those of proving that Clyde was a shark, a sperm whale, or an elephant. Yet, if the facts included the assertion that Clyde was an elephant, we ought then to be able to reason more efficiently, since we should be able, at least, to avoid subgoals such as an attempt to prove that Clyde was a shark. There is evidence that humans are able to perform this kind of reasoning task rapidly, without being overwhelmed by combinatorial considerations.

Some of the forward uses of delineations in taxonomic hierarchies can be efficiently built into the matcher without risking severe combinatorial problems. We shall describe how this might be done for some simple examples using the network formalism. Consider first the problem of trying to find a match for a goal arc _a_ between two fact nodes _N1_ and _N2_. We show this situation in Figure 29. If there is a fact arc _a_ between _N1_ and _N2_ (as shown by one of the dashed arcs in Figure 29), we then have an immediate match. We could restrict the matcher by permitting it to look only for such immediate matches. If none were found, we could explicitly apply production rules, such as the delineation shown in Figure 29, to solve the problem. Alternatively, we

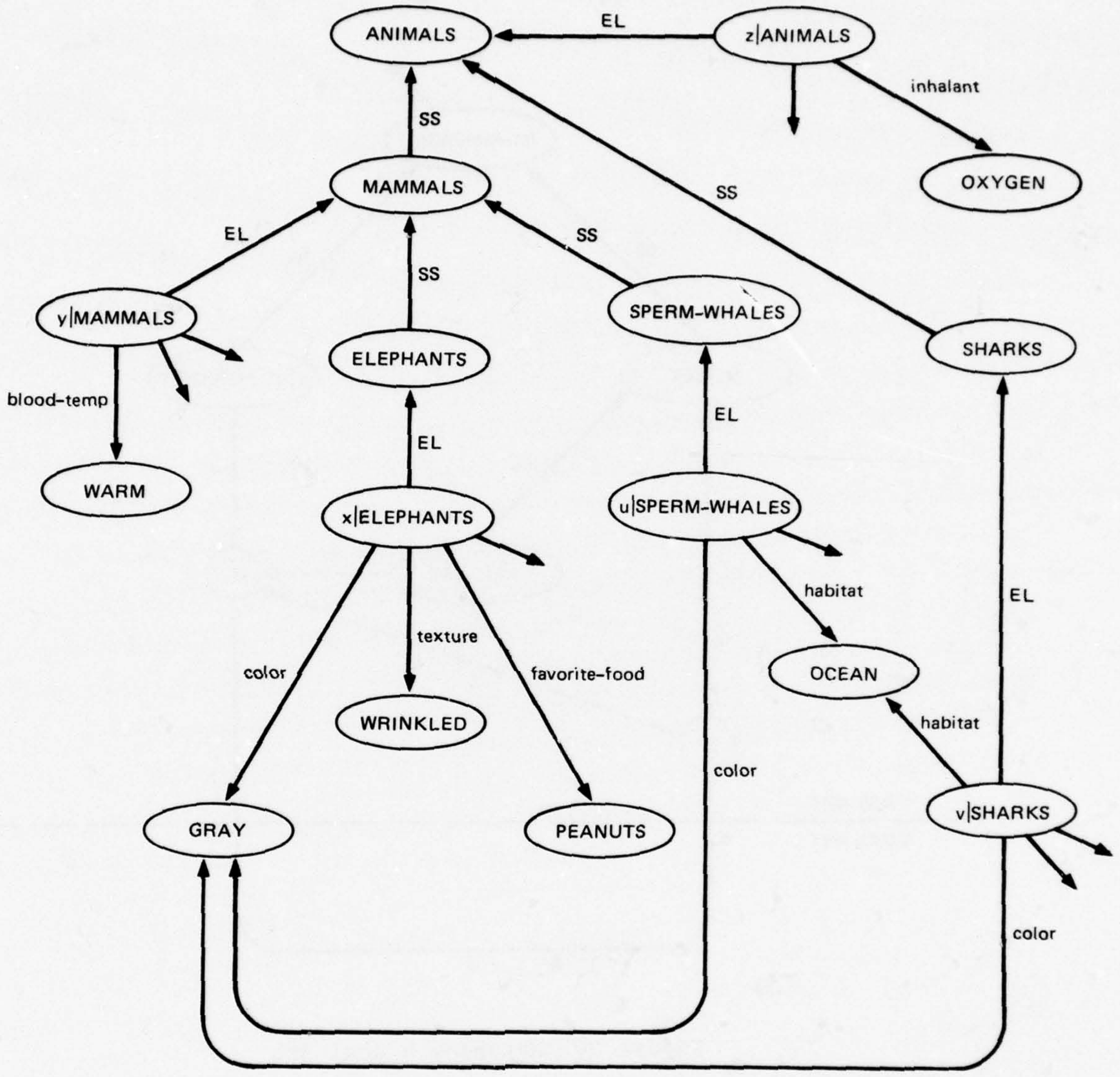

Figure 28 A Taxonomic Hierarchy of Sets and Their Delineations

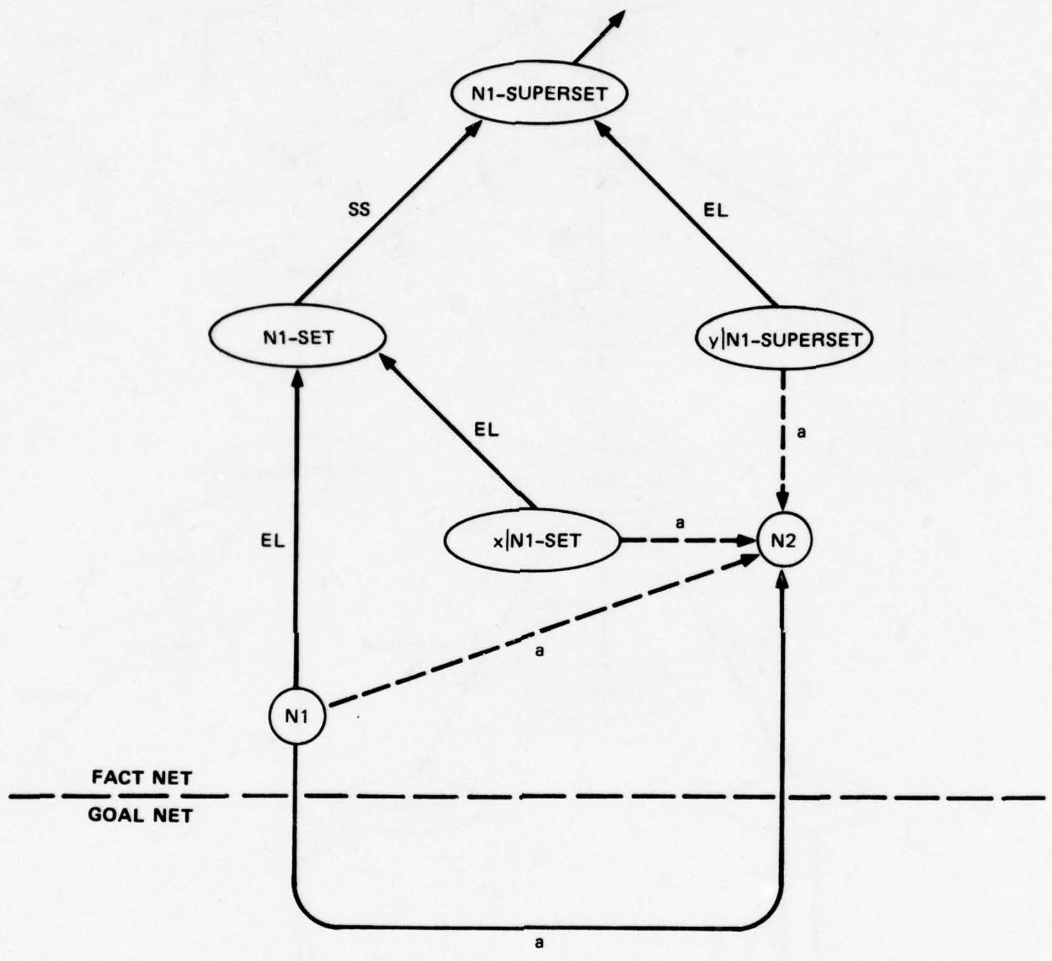

Figure 29 Matching a Goal Arc

could build this use of delineations right into the matcher. In taxonomic hierarchies that converge toward a small number of sets at the top, there is little harm in building into the matcher itself the ability to apply certain delineations in the forward direction.

As regards the example of Figure 29, if the matcher could not find an explicit _a_ arc in the fact network between _N1_ and _N2_, it would then ascend the taxonomic hierarchy from _N1_ -- checking for the presence of _a_ arcs to _N2_ from delineations of the sets (and supersets) to which _N1_ belongs. In Figure 29 we show by dashed arcs some of the possible _a_ arcs that the matcher is permitted to seek. If it can find such an arc, the match is successful. Unless all of the goal arcs can be matched, the matcher terminates with failure.

A system with this type of extended matcher operates as if an object automatically _inherited_ all the needed properties of its sets and supersets. In fact, the ease with which properties can be inherited is one of the advantages of using a structured-object formalism.

As an illustration of this process, let's consider the following examples based on Figure 28. First, suppose we want to prove that Clyde is gray when we know that Clyde is an elephant (but we don't know explicitly that Clyde is gray). This problem is represented in Figure 30 where we have included part of the net shown in Figure 28. Since there is no "color" arc within the fact net pointing from fact _CLYDE_ to fact _GRAY_, we cannot obtain an immediate match. So we move up to the ELEPHANTS delineation where we do have a "color" arc to fact _GRAY_. The matcher notes that CLYDE inherits this color arc and finishes with a successful match.

Next, suppose we want to prove that Clyde is warm-blooded when we know only that Clyde is an elephant. Again, we move up the taxonomic hierarchy to the delineation unit for MAMMALS where a match is readily determined.

Next, suppose we want to prove that Clyde breathes oxygen and is gray and warm-blooded, given only that Clyde is a mammal. An ascent

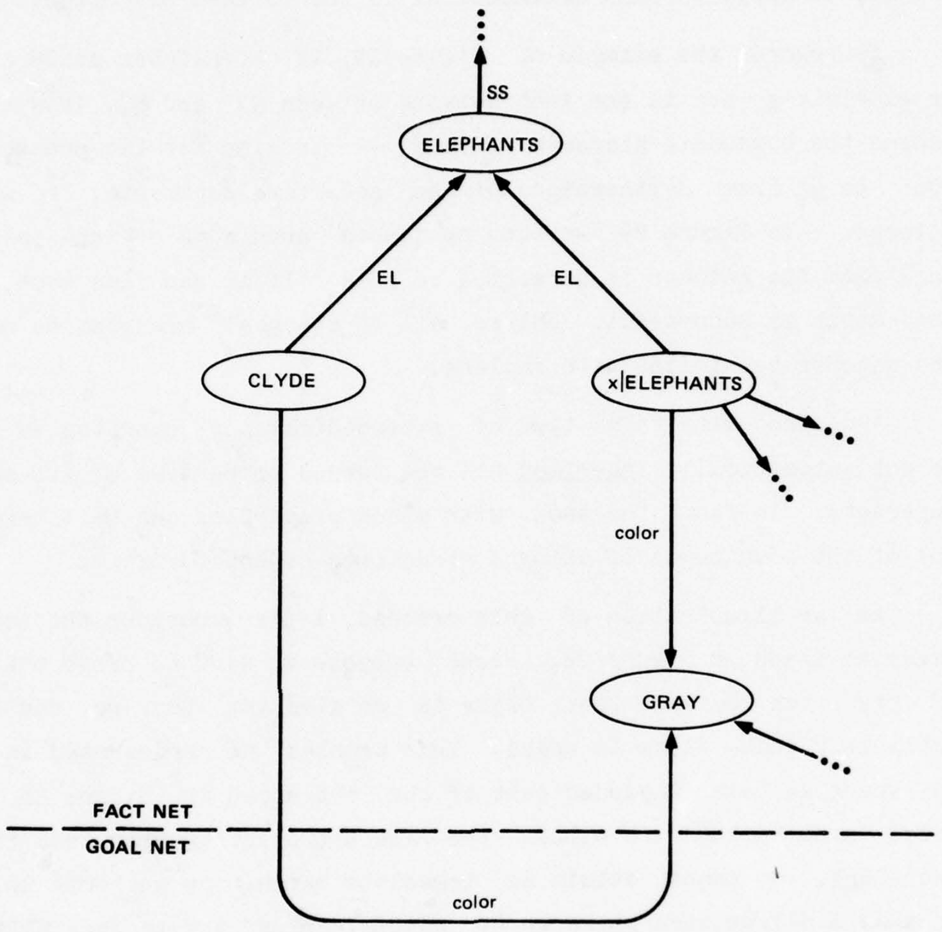

Figure 30 A Net for Proving that Clyde is Gray

along the delineation hierarchy picks up a "blood-temp" arc to WARM and an "inhalant" arc to OXYGEN, but not a complete match. These two properties are added explicitly to CLYDE before an attempt is made to prove the goal by rule-based means.

Strategies for matching a variable goal node against facts in the data base depend on the structure of the net. In the simplest case the variable goal node, say, x, will be tied to constant fact nodes, N1, N2, ..., Nk, by arcs labeled a1, a2, ..., ak, respectively. The situation will be as depicted in Figure 31. The constant nodes N1, ..., Nk will also have other incident arcs. Our attempt to find a match must look back through a1 arcs incident upon N1, a2 arcs incident upon N2, etc. (We assume that our implementation of the network makes it easy to trace through arcs in the "reverse" direction.) Some of these arcs will originate from constant nodes, some from delineations. A good strategy is to look first for a constant node, because the set of possible nodes in the fact net that might match x can be quite large if the delineations are considered. Suppose node Ni has the smallest set of constant nodes sending ai arcs to Ni. We attempt to match x against the nodes in this set and allow the matcher to use delineations in matching the other arcs.

In Figure 32, we show a simple example. In this case there is only one constant node, namely CLYDE, having the desired properties. In attempting a match against CLYDE, we must next find an EL arc between CLYDE and MAMMALS and a blood-temp arc between CLYDE and WARM. The first of these arcs is inferred by a subset chain, and the second is established by inheritance, so the match succeeds.

We will always be able to find at least one constant node to use as a candidate if we allow the matcher to look backward through SS and EL chains. Consider, for example, the problem shown in Figure 33. In this net, there is no "immediate" constant node to serve as a candidate match, but working down from MAMMALS through an SS and an EL chain puts us at the constant node, CLYDE. The rest of the match is easily handled by property inheritance. We can assume that a variable goal node will

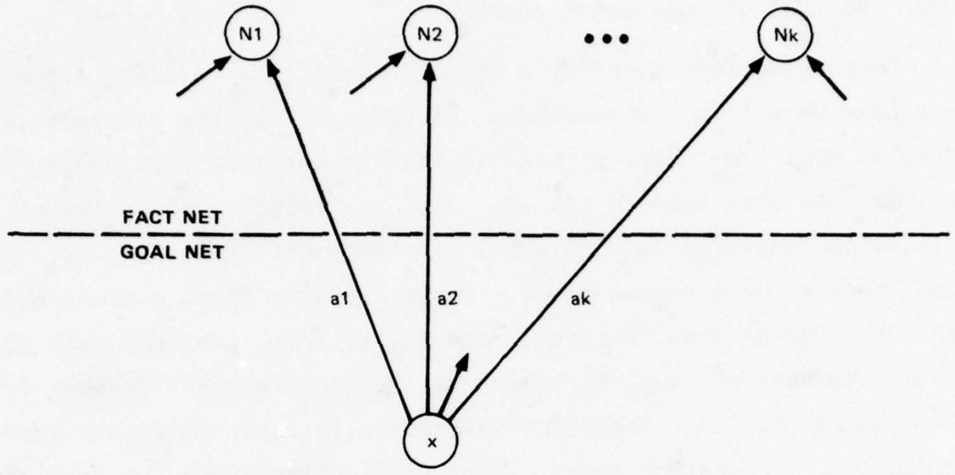

Figure 31 Matching a Variable Goal Node

always have an EL (or SS) arc pointing to something in the fact net. (We assume that every entity is at least a member of the universal set.)

The above process can be elaborated to deal with cases in which the goal net structure is more complex and contains more than one variable node. Each variable node must be properly matched for the entire goal structure to be matched. In any case, if no match can be obtained, either delineation rules must be used in the backward direction, or other rules must be applied to change the fact or the goal structures.

The problems of matching and property inheritance in structured representations have received a great deal of attention. Moore [13] has proposed a unification process that works with a type hierarchy. The SNIFFER system of Fikes and Hendrix [40] embodies ideas similar to those described above. Fahlman [41] has recently proposed that structured representations be embedded in hardware, so that parallel searches can be conducted.

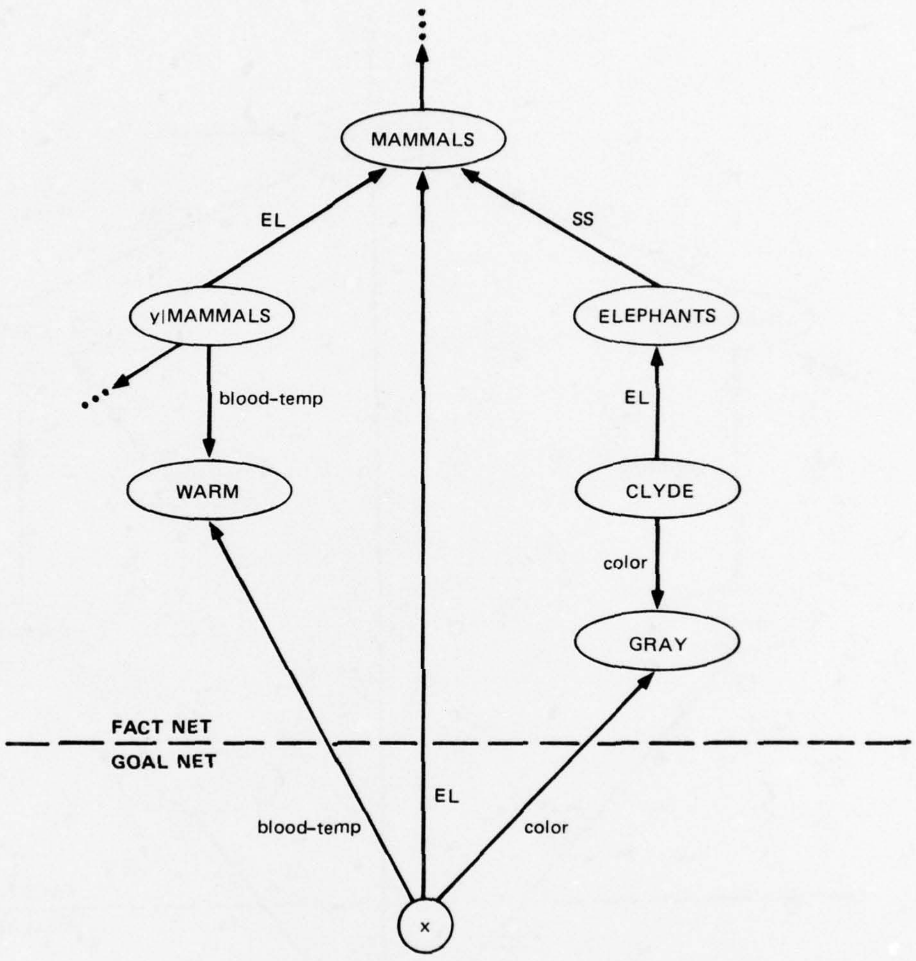

Figure 32 An Example with a Variable Goal Node

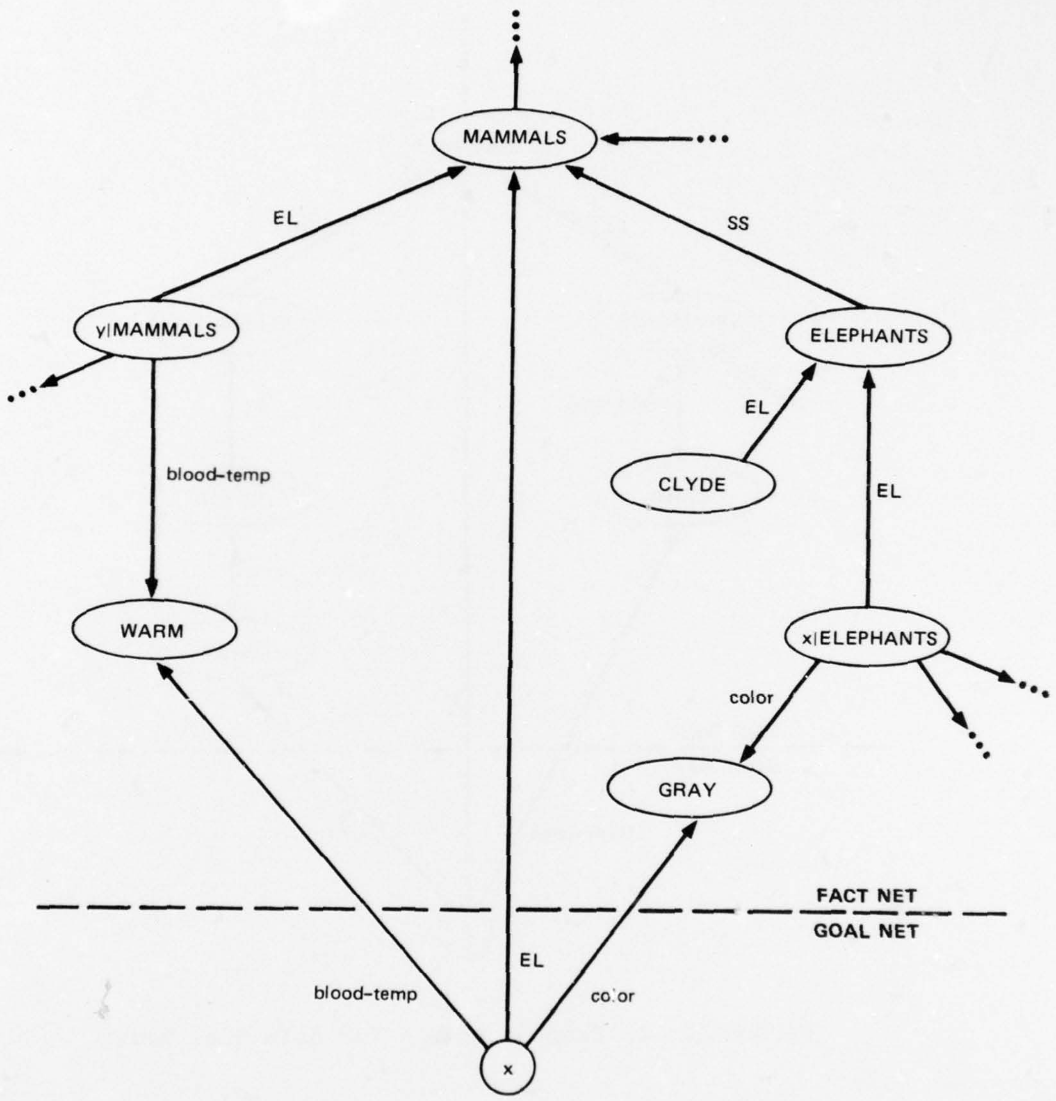

Figure 33 Another Example with a Variable Goal Node

F. Net Rules

Earlier, we mentioned the use of enclosures to represent network implications. These implications can be used as forward or backward rules in semantic network-based production systems. For example, the implication

```
    {EL(x, DEPARTMENTS) & EQ[manager(x), y]}
                => EQ[works-in(y), x]
```

might be represented by the network structure shown in Figure 34.

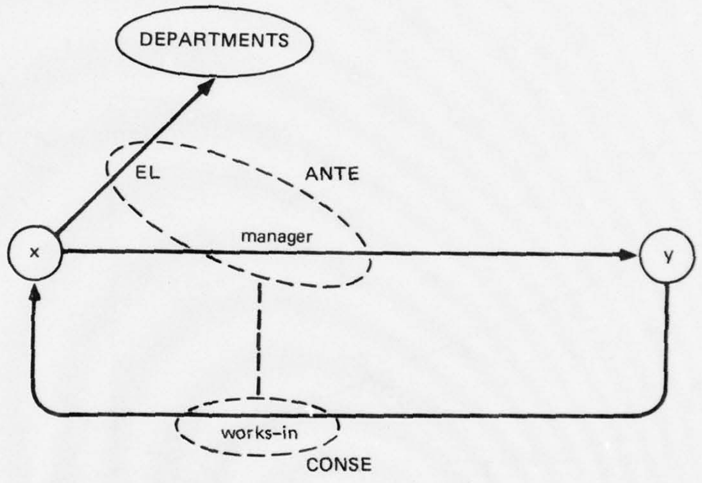

Figure 34 Representing an Implication

To use a network implication as a forward rule, the ANTE structure (regarded as a goal) must match existing network fact structures. The CONSE structure (appropriately instantiated) can then be added to the fact network. To use a network implication as a backward rule, the CONSE structure (regarded as a fact) must match the goal structure. The ANTE structure (appropriately instantiated) is then the subgoal produced by the rule application. Here too the situation is more complex (involving AND/OR trees and substitution consistency testing) when the goal structure is first broken into component structures and these are matched individually by CONSE structures of rules.

As a more complex example, we illustrate in Figure 35 the network version of the following implication:

```
EQ[y, wife-of(x)]  =>
                       {EL[m(x,y), MARRIAGE-EVENTS]
                         &  EQ[x, male(m(x,y))]
                         &  EQ[y, female(m(x,y))]}
```

The node labeled, m(x,y) is a Skolem function node. Every forward application of the rule in Figure 35 creates a newly instantiated m(x,y) node.

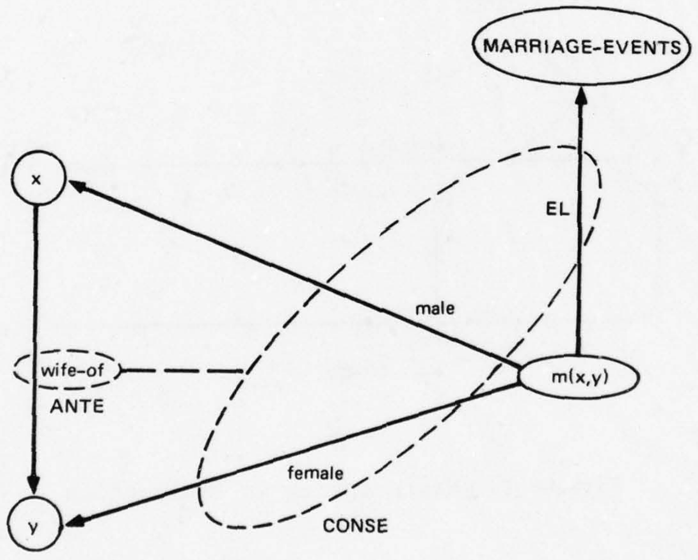

Figure 35 A Network Implication with a Skolem-Function

G. Defaults and Contradictory Information

Many descriptive statements of the form "all x's have property P" must be regarded as only approximately true. Perhaps most x's do have property P, but typically we will come across exceptions. Examples of this kind of exception abound: all birds can fly (except ostriches), all men have two legs (except certain amputees), all lemons are yellow (except unripe ones and mutant orange ones), etc. It appears that many general synthetic (as opposed to analytic or definitional) statements that we might make about the world are incorrect unless qualified. Furthermore, the qualifications probably have to be so numerous that the formalism would become unmanageably cumbersome if we attempted to include them all explicitly. Is there a way around this difficulty that still preserves the simplicity of a predicate calculus language?

One approach is to weaken universal quantification by allowing certain implicit exceptions. Thus, the statement "all elephants are gray" might initially be given without listing any exceptions. Such a statement would allow us, upon learning that Clyde is an elephant, to deduce that he is gray. Later, if we learn that Clyde is actually white, we must retract our deduction about his grayness and change the universal statement about elephants so that it excludes Clyde. After this change is made, it will no longer be possible to deduce erroneous conclusions about Clyde's color.

The way in which the matcher uses property inheritance provides an automatic mechanism for dealing with exceptions like this. The matcher uses inheritance to deduce a property of an object from a delineation of its class only if specific information about the property of that object is lacking. Suppose, for example, that we want to know the color of Clyde. Such a query might be represented by the goal network shown in Figure 36.

In answering this query, we first attempt a direct match with the fact network. Suppose our fact network states that Clyde's color is white. In this case, the match substitution is {WHITE/x}, and we obtain the response "white" directly.

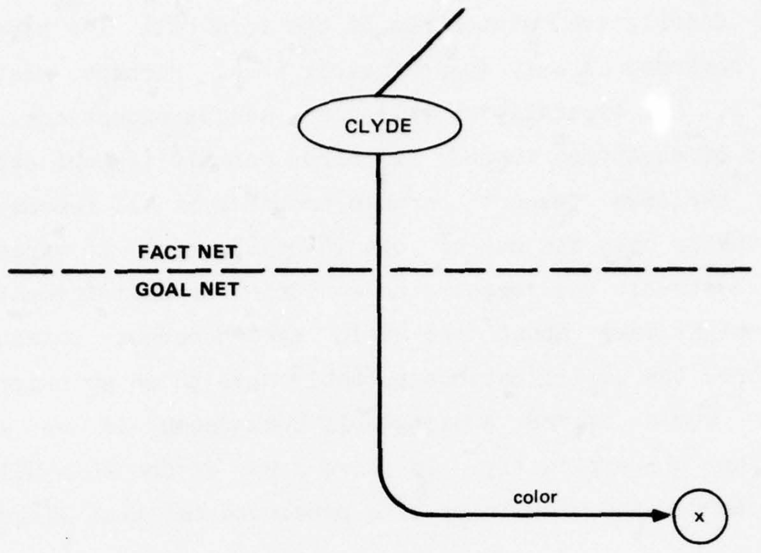

Figure 36 A Goal Net

If our fact network states only that Clyde is an elephant, the matcher automatically uses the delineation of ELEPHANTS to answer our query. Such a delineation might state that all elephants are gray, and thus our answer would be that Clyde is gray.

This scheme for countermanding general information by contradictory specific information can be extended to several hierarchical levels. For such a scheme to work, the use of delineations to deduce properties must always proceed from the more specific to the more general. With this built-in ordering of matching and retrieval processes, information at the more specific levels protects the system from making possibly contradictory deductions based on higher-level delineations. It is as if the universal quantifiers of delineations specifically exclude from their domains all of the more specific objects that would contradict the delineation.

Schemes of this sort do entail certain problems, however. Suppose, for example, that an object in the taxonomic hierarchy belongs to two different sets and that the delineations of these sets are contradictory. We show an example of this in Figure 37. In that illustration, we do not show an explicit color arc for CLYDE, but CLYDE inherits contradictory color values (assuming that ~EQ(GRAY, WHITE)). One way to deal with this problem is to indicate something about the quality of each arc in a delineation. In our example, if the color arc in the ALBINOS delineation were to dominate the color arc in the ELEPHANTS delineation, we would then always attempt to inherit the color value from the ALBINOS delineation first.

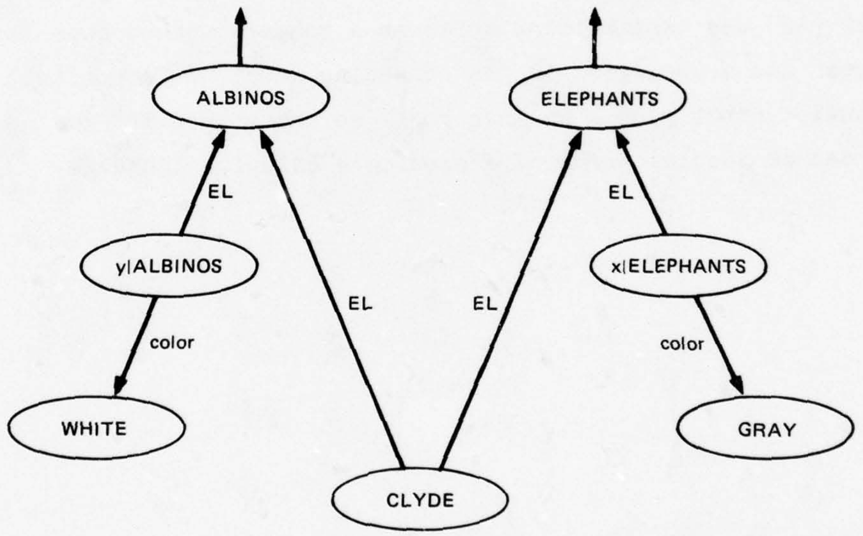

Figure 37 A Net with Contradictory Delineations

A common way to indicate that an arc of a delineation is of low priority is to mark it as a default. Default delineations can be used only if there is no other way to derive the needed information. In general, though, we would need an ordering of the default markers. If both delineations in Figure 37 were marked simply as defaults, for example, we would find ourselves at an impasse. We could prove that

Clyde was gray if we could not prove that he was any other color. But, by the same token we could also prove that he was another color, namely white, in the absence of proof that he was any other color except white. And so on.

We must also be equally careful about using default delineations as forward-rule applications, because we might then add objects to the fact data base that contradict existing or subsequent specific facts. The forward use of delineations must be coupled with "truth maintenance" techniques to ensure that contradictory facts (and facts that might be derived from them) are either purged or otherwise inactivated.

Semantic networks of one sort or another find several applications in AI systems. Fikes and Hendrix [40] have developed a deduction system based on semantic networks for use in natural-language processing. Duda et al, [42] use semantic networks in a computer-based consulting system that can aid a geologist in his reasoning about ore deposits. It is our contention that such uses can best be understood if the networks are regarded as special forms of a predicate calculus language.

VII Conclusions

In this report we have described how elaborations of production systems are used with predicate calculus representations in systems for problem solving. Our thesis is that these abstract systems and representational formalisms provide a useful common foundation for describing, understanding, and improving actual AI systems across a wide range of different applications. To be sure, real AI programs are considerably more elaborate than the systems we have described here. But most of the basic mechanisms seem to be encompassed by our various production systems.

The fundamental ideas discussed in this report can be usefully expanded in several ways. One important development involves "layers" of production systems. Comprising the first (or object) layer is the main production system with its global data base and rule set. Its control strategy, however, is implemented as a higher-level (or meta) production system with its own global data base and rule set. (Davis [43], Shrobe [44], and McDermott [45] have all discussed layered systems of this type.) The work of Weyhrauch [46] on interreferencing is particularly important in this regard.

Another essential area for future work to focus upon is the use of production systems operating concurrently and in parallel with partially shared global data bases. Communication among the systems then becomes a matter of particular importance. Hewitt [47] and Lesser and Corkill [48] are exploring these kinds of systems. In this report we have largely been concerned with the structure and operation of AI systems more or less divorced from the content of the problem information utilized by them. Except for some extremely simple examples, we have not touched on the vital issue of how certain types of complex information are to be represented in the predicate calculus (or in any other formalism). AI researchers are still in doubt about the best way to represent certain complex concepts, such as knowledge, beliefs, goals, and actions. Many researchers agree that some sort of predicate calculus formalism for these concepts is needed, but there is still disagreement as to which type would be both adequate and efficient. Whatever the form of these yet-to-be-devised representations, it nevertheless seems clear that the basic production system architecture for AI systems will remain extremely important.

REFERENCES

1. E. H. Shortliffe, Computer-Based Medical Consultations: MYCIN, American Elsevier, New York, 1976.

2. H. E. Pople, Jr., "The Formation of Composite Hypotheses in Diagnostic Problem Solving: An Exercise in Synthetic Reasoning," Proc. 5th Intl. Joint Conf. on AI, Aug. 1977, pp 1030-1037.

3. R. M. Stallman and G. J. Sussman, "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis," Artificial Intelligence, vol. 9, no. 2, Oct. 1977, pp 135-196.

4. W. W. Bledsoe, "Non-Resolution Theorem Proving," Artificial Intelligence, vol. 9, no. 1, Aug. 1977, pp 1-35.

5. D. Sagalowicz, "IDA: An Intelligent Data Access Program, Proc. 3rd Intl. Conf. on Very Large Data Bases, Tokyo, Japan, Oct. 1977.

6. D. E. Walker, ed., Understanding Spoken Language, North-Holland, New York, 1978.

7. G. G. Hendrix et. al., "Developing a Natural Language Interface to Complex Data," ACM Trans. on Database Systems, vol. 3, no. 2, June 1978, pp 105-147.

8. D. Waltz, "Natural Language Access to a Large Data Base: An Engineering Approach," Proc. 4th Intl. Joint Conf. on AI, Sept. 1975, pp 868-872.

9. R. Davis and J. King, "An Overview of Production Systems," in Machine Intelligence 8, E. Elcock and D. Michie, eds., Wiley, New York, 1976, pp 300-332.

10. N. J. Nilsson, Problem-Solving Methods in Artificial Intelligence, McGraw-Hill Book Co., New York, 1971

11. B. Raphael, The Thinking Computer: Mind Inside Matter, W. H. Freeman and Company, San Francisco, 1976.

12. D. Bobrow and B. Raphael, "New Programming Languages for Artificial Intelligence Research," ACM Computing Surveys, vol. 6, 1974, pp 153-174.

13. R. Moore, Reasoning from Incomplete Knowledge in a Procedural Deduction System, MIT Artificial Intelligence Lab, AI-TR-347, 1975.

14. H. Gallaire and J. Minker, eds., Logic and Data Bases, Plenum, New York, 1978.

15. R. Kowalski, Logic for Problem Solving, Dept. of Computational Logic, School of Artificial Intelligence, University of Edinburgh, Edinburgh Scotland, Memo 70, 1974.

16. S. Sickel, "A Search Technique for Clause Interconnectivity Graphs, IEEE Trans. Comput., C-25, 1976, pp 823-835.

17. P. Klahr, "Planning Techniques for Rule Selection in Deductive Question Answering," in Pattern-Directed Inference Systems, D. Waterman and F. Hayes-Roth, eds., Academic Press, New York, 1978, pp. 223-239.

18. P. Cox, Deduction Plans: A Graphical Proof Procedure for the First-Order Predicate Calculus, Dept. of Computer Science, University of Waterloo, Research Report CS-77-28, 1977.

19. N. J. Nilsson, A Production System for Automatic Deduction, SRI Artificial Intelligence Center Tech. Note 148, July 1977.

20. Z. Manna and R. Waldinger, A Deductive Approach to Program Synthesis SRI Artificial Intelligence Center Tech. Note 177, Dec. 1978.

21. D. Warren et. al., "PROLOG: The Language and its Implementation Compared with LISP," Proc. Symp. on AI and Programming Languages, SIGART Newsletter, no. 64, Aug. 1977.

22. R. E. Fikes and N. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," Artificial Intelligence, vol. 2, 1971, pp. 189-208.

23. C. Dawson and L. Siklossy, "The Role of Preprocessing in Problem-Solving Systems, Proc. of 5th Intl. Joint Conf. on AI, 1977, pp. 465-471.

24. R. Waldinger, Achieving Several Goals Simultaneously, SRI Artificial Intelligence Center Tech. Note 107, 1975.

25. A. Newell et al, "Report on a General Problem-Solving Program for a Computer," Proc. Intl. Conf. Information Processing, UNESCO, Paris, 1960, pp 256-264.

26. C. Rieger and P. London, "Subgoal Protection and Unravelling During Plan Synthesis," Proc. 5th Intl. Joint Conf. on AI, 1977, pp. 487-493.

27. G. Sussman, A Computational Model of Skill Acquisition, American Elsevier, New York, 1975.

28. E. Sacerdoti, A Structure for Plans and Behavior, American Elsevier, New York, 1977.

29. A. Tate, "Generating Project Networks," Proc. 5th Intl. Joint Conf. on AI, 1977, pp 888-893.

30. E. Sacerdoti, "Planning in a Hierarchy of Abstraction Spaces," Artificial Intelligence, vol. 5, 1974, pp 115-135.

31. C. Green, The Application of Theorem Proving to Question-Answering Systems, Stanford Artificial Intelligence Project Memo AI-96, 1969.

32. M. Minsky, "A Framework for Representing Knowledge," in The Psychology of Computer Vision, P. Winston, ed., McGraw-Hill, New York, 1975.

33. R. Schank and R. Abelson, Scripts, Plans, Goals and Understanding Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1977.

34. M. Stefik, "An Examination of a Frame-Structured Representation System," Stanford Heuristic Programming Project Memo HPP-78-13, (Working Paper), Sept. 1978.

35. D. Bobrow and T. Winograd, "An Overview of KRL, A Knowledge Representation Language," Cognitive Science, vol. 1, no. 1, Jan. 1977.

36. R. Roberts and I. Goldstein, "The FRL Primer," MIT AI Lab Memo no. 408, July 1977.

37. G. Hendrix, "Encoding Knowledge in Partitioned Networks," SRI AI Center Tech. Note 164, June 1978. (To be published in Associative Networks--The Representation and Use of Knowledge in Computers, N. Findler, ed., Academic Press, New York.)

38. A. Deliyanni and R. Kowalski, "Logic and Semantic Networks," in Logic and Data Bases, H. Gallaire and J. Minker, eds., Plenum, New York, 1978.

39. M. Kay, "The Mind System," in Natural Language Processing, R. Rustin, ed., Algorithmics Press, New York, 1973, pp. 153-188.

40. R. Fikes and G. Hendrix, "A Network-Based Knowledge Representation and its Natural Deduction System," Proc. 5th Intl. Joint Conf. on AI, 1977, pp 235-246.

41. S. Fahlman, A System for Representing and Using Real-World Knowledge, MIT Ph.D. Dissertation, Sept. 1977.

42. R. Duda et al, "Semantic Network Representations in Rule-Based Inference Systems," Pattern-Directed Inference Systems, D. Waterman and F. Hayes-Roth, eds., Academic Press, New York, 1978.

43. R. Davis, Application of Meta-Level Knowledge to the Construction, Maintenance and Use of Large Knowledge Bases, Stanford AI Lab. Memo AIM-283, 1976.

44. H. Shrobe, "Explicit Control of Reasoning in the Programmer's Apprentice," Proc. 4th Workshop on Automated Deduction, Austin, Texas, Feb. 1979, pp. 97-102.

45. D. McDermott, "A Deductive Model of Control of a Problem Solver," Proc. Workshop Pattern-Directed Inference Systems, SIGART Newsletter 63, 1977, pp. 2-7.

46. R. Weyhrauch, Prolegomena to a Theory of Mechanized Formal Reasoning, forthcoming Stanford AI Lab Memo, 1979.

47. C. Hewitt, "Viewing Control Structures as Patterns of Passing Messages," Artificial Intelligence, vol. 8, no. 3, June 1977, pp. 323-364.

48. V. Lesser and D. Corkill, Cooperative Distributed Problem Solving, Computer and Info. Science Dept. Report 78-7, University of Massachusetts, 1978.